
大数据科学丛书
Big Data Science



Spark
内核机制解析及性能调优

王家林　段智华　等编著






机械工业出版社
CHINA MACHINE PRESS

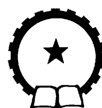
VISIT…

LANZAROTE Caliente.com

大数据科学丛书

Spark 内核机制解析及性能调优

王家林　段智华　等编著



机 械 工 业 出 版 社

Spark 建立在抽象的 RDD 之上，要理解 Spark，就需要理解 RDD。本书详细讲述了 RDD 的功能及内部实现的机制，通过剖析源代码引导读者深入理解 Spark 集群部署的内部机制、Spark 内部调度机制、Executor 的内部机制和 Shuffle 的内部机制，进而讲述了“钨丝计划”的内部机制。本书的最后一章是任何 Spark 应用者都非常关注的 Spark 性能调优内容。通过学习本书，可以使读者对 Spark 内核有更加深入的理解，从而实现对 Spark 系统深度调优、Spark 生产环境下故障的定位和排除，以及 Spark 的二次开发和系统高级运维。

本书适合于对大数据开发有兴趣的在校学生。同时，对于有分布式计算框架应用经验的人员，本书也可以作为 Spark 源代码解析的参考书籍。

图书在版编目（CIP）数据

Spark 内核机制解析及性能调优/王家林等编著．—北京：机械工业出版社，2016.10

（大数据科学丛书）

ISBN 978－7－111－55442－4

Ⅰ.①S…　Ⅱ.①王…　Ⅲ.①数据处理软件　Ⅳ.①TP274

中国版本图书馆 CIP 数据核字（2016）第 278303 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）

责任编辑：王　斌

责任校对：张艳霞

印刷（　　　装订）

2017 年 1 月第 1 版・第 1 次印刷

184mm×260mm・22 印张・537 千字

0001－3000 册

标准书号：ISBN 978－7－111－55442－4

定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

服务咨询热线：（010）88379833

读者购书热线：（010）88379649

封面无防伪标均为盗版

网络服务

机 工 官 网：www.cmpbook.com

机 工 官 博：weibo.com/cmp1952

教育服务网：www.cmpedu.com

金 书 网：www.golden－book.com

前　言



起源于美国加州大学伯克利分校 AMP 实验室的 Spark 是当今大数据领域最活跃、最热门的大数据计算处理框架，2009 年 Spark 诞生于 AMP 实验室，2010 年 Spark 正式成为开源项目，2013 年 Spark 成为 Apache 基金项目，2014 年 Spark 成为 Apache 基金的顶级项目。Spark 成功构建了一体化、多元化的大数据处理体系，成功使用 Spark SQL、Spark Streaming、MLlib、GraphX 解决了大数据领域的 Batch Processing、Stream Processing、Adhoc Query 等核心问题，Spark SQL、Spark Streaming、Mllib、GraphX 四个子框架和 Spark 核心库之间互相共享数据及相互操作，Spark 生态系统强大的集成能力是其他大数据平台无可匹敌的。

本书主要面向的对象是广大的 Spark 爱好者和大数据开发者，以 Spark 内核解析及性能调优为主导，由浅入深，对 Spark 内核运行机制从源代码角度加以详细解析，全书共分 9 章，分别是：RDD 的功能解析、RDD 的运行机制、部署模式（Deploy）解析、Spark 调度器（Scheduler）运行机制、执行器（Executor）、Spark 的存储模块（Storage）、Shuffle 机制、钨丝计划（Project Tungsten）以及性能优化。读者通过对这些内容的深入学习，将能够较为透彻地掌握 Spark 这一大数据计算框架的应用方法。

参与本书编写的有王家林、段智华、张敏等。

在本书阅读过程中，如发现任何纰漏或有任何疑问，可以加入本书的阅读群（QQ：284078981）提出问题，会有专人答疑。同时，该群也会提供本书所用案例源代码。

如果读者想要了解或者学习更多大数据相关技术，可以关注 DT 大数据梦工厂微信公众号 DT_Spark 及 QQ 群 284078981，或者扫描下方二维码咨询，也可以通过 YY 客户端登录 68917580 永久频道直接体验。

王家林老师的新浪微博是 http：//weibo. com/ilovepains/，欢迎大家在微博上与作者进行互动。

由于时间仓促，书中难免存在不妥之处，请读者谅解，并提出宝贵意见。



王家林 2016. 10. 8 日于深圳

目　　录

前言
第 1 章　RDD 功能解析 …… 1
1.1　RDD 产生的技术背景及功能 …… 1
1.2　RDD 的基本概念 …… 2
1.2.1　RDD 的定义 …… 2
1.2.2　RDD 五大特性 …… 5
1.2.3　RDD 弹性特性的 7 个方面 …… 7
1.3　创建 RDD 的方式 …… 13
1.3.1　通过已经存在的 Scala 集合创建 RDD …… 13
1.3.2　通过 HDFS 和本地文件系统创建 RDD …… 13
1.3.3　其他的 RDD 的转换 …… 14
1.3.4　其他的 RDD 的创建 …… 20
1.4　RDD 算子 …… 21
1.5　RDD 的 Transformation 算子 …… 24
1.5.1　Transformation 的定义 …… 24
1.5.2　Transformation 在 RDD 中的角色定位及功能 …… 24
1.5.3　Transformation 操作的 Lazy 特性 …… 24
1.5.4　通过实践说明 Transformation 的 Lazy 特性 …… 25
1.6　RDD 的 Action 算子 …… 25
1.6.1　Action 的定义 …… 25
1.6.2　Action 在 RDD 中的角色定位及功能 …… 25
1.7　小结 …… 27
第 2 章　RDD 的运行机制 …… 28
2.1　RDD 依赖关系 …… 28
2.1.1　窄依赖（Narrow Dependency） …… 28
2.1.2　宽依赖（Shuffle Dependency） …… 30
2.2　有向无环图（Directed Acyclic Graph，DAG） …… 31
2.2.1　什么是 DAG …… 31
2.2.2　DAG 的生成机制 …… 32
2.2.3　DAG 的逻辑视图 …… 33
2.3　RDD 内部的计算机制 …… 34
2.3.1　RDD 的计算任务（Task） …… 34
2.3.2　RDD 的计算过程 …… 35

2.4 RDD 中缓存的适用场景和工作机制 …… 41
2.4.1 缓存的使用 …… 41
2.4.2 缓存的适用场景 …… 42
2.4.3 缓存工作机制解析 …… 43
2.5 RDD 的检查点（Checkpoint）的适用场景和工作机制 …… 45
2.5.1 Checkpoint 的触发 …… 45
2.5.2 Checkpoint 的适用场景 …… 46
2.5.3 Checkpoint 工作机制解析 …… 46
2.6 RDD 容错原理及其四大核心要点 …… 49
2.6.1 RDD 容错原理 …… 49
2.6.2 RDD 容错的四大核心要点 …… 49
2.7 通过 WordCount 实践 RDD 内部机制 …… 51
2.7.1 WordCount 案例实践 …… 51
2.7.2 解析 RDD 生成的内部机制 …… 53
2.8 小结 …… 54
第 3 章 部署模式（Deploy）解析 …… 55
3.1 部署模式概述 …… 55
3.2 应用程序的部署 …… 55
3.2.1 应用程序部署的脚本解析 …… 55
3.2.2 应用程序部署的源代码解析 …… 58
3.3 Local 与 Local－Cluster 部署 …… 69
3.3.1 Local 部署 …… 70
3.3.2 Local[*]与 Local[N]部署 …… 70
3.3.3 Local[* ,M]与 Local[N,M]部署 …… 70
3.3.4 Local－Cluster[S,C,M]部署 …… 72
3.4 Spark Standalone 部署 …… 72
3.4.1 部署框架 …… 72
3.4.2 应用程序的部署 …… 73
3.4.3 Master 的部署 …… 84
3.4.4 Worker 的部署 …… 98
3.4.5 内部交互的消息机制 …… 108
3.4.6 Master HA 的部署 …… 110
3.5 Spark on YARN 的部署模型 …… 115
3.5.1 部署框架 …… 115
3.5.2 应用程序的部署 …… 118
3.6 小结 …… 124
第 4 章 Spark 调度器（Scheduler）运行机制 …… 125
4.1 Spark 运行的核心概念 …… 125

4.1.1 Spark 运行的基本对象 …… 125
4.1.2 Spark 运行框架及各组件的基本运行原理 …… 126
4.2 Spark Driver Program 剖析 …… 127
4.2.1 什么是 Spark Driver Program …… 127
4.2.2 SparkContext 原理剖析 …… 128
4.2.3 SparkContext 源代码解析 …… 129
4.3 Spark Job 的触发 …… 134
4.3.1 Job 的逻辑执行（General Logical Plan） …… 134
4.3.2 Job 具体的物理执行 …… 135
4.3.3 Job 触发流程源代码解析 …… 138
4.4 高层的 DAG 调度器（DAGScheduler） …… 140
4.4.1 DAG 的定义 …… 140
4.4.2 DAG 的实例化 …… 140
4.4.3 DAGScheduer 划分 Stage 的原理 …… 142
4.4.4 DAGScheduer 划分 Stage 的具体算法 …… 143
4.4.5 Stage 内部 Task 获取最佳位置的算法 …… 148
4.5 底层的 Task 调度器（TaskScheduler） …… 150
4.5.1 TaskScheduer 原理剖析 …… 151
4.5.2 TaskScheduer 源代码解析 …… 152
4.6 调度器的通信终端（SchedulerBackend） …… 157
4.6.1 SchedulerBackend 原理 …… 157
4.6.2 SchedulerBackend 源代码解析 …… 158
4.6.3 Spark 程序的注册机制 …… 160
4.6.4 Spark 程序对计算资源 Executor 的管理 …… 163
4.7 小结 …… 167
第 5 章 执行器（Executor） …… 168
5.1 Executor 的创建、分配、启动及异常处理 …… 169
5.1.1 Executor 的创建 …… 169
5.1.2 Executor 的资源分配 …… 177
5.1.3 Executor 的启动 …… 183
5.1.4 Executor 的异常处理 …… 188
5.2 执行器的通信接口（ExecutorBackend） …… 190
5.2.1 ExecutorBackend 接口与 Executor 的关系 …… 190
5.2.2 ExecutorBackend 的不同实现 …… 191
5.2.3 ExecutorBackend 中的通信 …… 194
5.3 执行器（Executor）中任务的执行 …… 198
5.3.1 Executor 中任务的加载 …… 198
5.3.2 Executor 中的任务线程池 …… 199

5.3.3 任务执行失败处理 …… 199
5.3.4 剖析 TaskRunner …… 201
5.4 小结 …… 202
第 6 章 Spark 的存储模块（Storage） …… 203
6.1 Storage 概述 …… 203
6.1.1 Storage 的概念 …… 203
6.1.2 Storage 的设计模式 …… 204
6.2 Storage 模块整体架构 …… 204
6.2.1 通信层 …… 205
6.2.2 存储层 …… 208
6.2.3 Partition 与 Block 的对应关系 …… 235
6.3 不同 Storage Level 对比 …… 236
6.4 Executor 内存模型 …… 237
6.5 Tachyon …… 239
6.5.1 Tachyon 简介 …… 239
6.5.2 Tachyon API 的使用 …… 240
6.5.3 Tachyon 在 Spark 中的使用 …… 242
6.6 小结 …… 245
第 7 章 Shuffle 机制 …… 246
7.1 Shuffle 概述 …… 246
7.2 Shuffle 的框架 …… 248
7.2.1 Shuffle 的框架演进 …… 248
7.2.2 Shuffle 的框架内核 …… 249
7.2.3 Shuffle 框架的源代码解析 …… 250
7.2.4 Shuffle 的注册 …… 254
7.2.5 Shuffle 读写数据的源代码解析 …… 255
7.3 基于 Hash 的 Shuffle …… 261
7.3.1 基于 Hash 的 Shuffle 内核 …… 261
7.3.2 基于 Hash 的 Shuffle 写数据的源代码解析 …… 265
7.4 基于 Sort 的 Shuffle …… 270
7.4.1 基于 Sort 的 Shuffle 内核 …… 271
7.4.2 基于 Sort 的 Shuffle 写数据的源代码解析 …… 273
7.5 基于 Tungsten Sort 的 Shuffle …… 279
7.5.1 基于 Tungsten Sort 的 Shuffle 内核 …… 279
7.5.2 基于 Tungsten Sort 的 Shuffle 写数据的源代码解析 …… 281
7.6 小结 …… 286
第 8 章 钨丝计划（Project Tungsten） …… 287
8.1 钨丝计划（Project Tungsten）概述 …… 287



8.2 内存管理模型 …… 288
8.2.1 现有内存管理的机制 …… 288
8.2.2 Project Tungsten 内存管理的模型及其源代码的解析 …… 290
8.3 基于内存管理模型的 Shuffle 二进制数据处理 …… 305
8.3.1 插入记录时二进制数据的处理 …… 308
8.3.2 spill 时二进制数据的处理 …… 312
8.4 小结 …… 313
第 9 章 性能优化 …… 314
9.1 Spark 的配置机制 …… 314
9.1.1 通过 SparkConf 配置 Spark …… 314
9.1.2 通过 spark - submit 配置 Spark …… 315
9.1.3 通过配置文件配置 Spark …… 316
9.1.4 Spark 配置机制总结 …… 316
9.2 性能诊断 …… 317
9.2.1 WebUI 的 8080 端口 …… 317
9.2.2 WebUI 的 18080 端口 …… 318
9.2.3 WebUI 的 4040 端口 …… 319
9.2.4 WebUI 的 Jobs 页面 …… 319
9.2.5 WebUI 的 Stages 页面 …… 320
9.2.6 WebUI 的 Storage 页面 …… 321
9.2.7 WebUI 的 Environment 页面 …… 322
9.2.8 WebUI 的 Executors 页面 …… 324
9.2.9 Driver 和 Executor 的日志 …… 324
9.3 性能优化 …… 325
9.3.1 程序编写准则 …… 325
9.3.2 并行度 …… 330
9.3.3 资源参数调优 …… 331
9.3.4 序列化与压缩 …… 332
9.3.5 内存调优 …… 334
9.3.6 广播大变量 …… 336
9.3.7 持久化与 checkpoint …… 337
9.3.8 数据本地性 …… 341
9.3.9 垃圾回收调优 …… 342
9.3.10 Shuffle 调优 …… 343
9.4 小结 …… 344

第 1 章　RDD 功能解析

Spark 建立在抽象的 RDD（Resilient Distributed Datasets，弹性分布式数据集）之上，使得它可以用一致的方式处理大数据不同的应用场景。Spark 把所有需要处理的数据转化成为 RDD，然后对 RDD 进行一系列的算子运算，从而得到结果。RDD 是一个容错的、并行的数据结构，它可以将数据存储到内存和磁盘中，并能控制数据分区，且提供了丰富的 API 来操作数据。Spark 一体化多元化的解决方案极大地减少了开发和维护的人力成本，以及部署平台的物力成本，并在性能方面有极大的优势，特别适合于迭代计算，如机器学习和图计算；同时 Spark 对 Scala 和 Python 交互式 Shell 的支持也极大地方便了通过 Shell 直接使用 Spark 集群来验证解决问题的方法，这对于原型开发至关重要，对数据分析人员有着无法抗拒的吸引力。本章将详细介绍 RDD 的功能。

Section 1.1　RDD 产生的技术背景及功能

Hadoop 的 MapReduce 是一种基于数据集的工作模式，这种模式的工作方式是：从物理存储上加载数据，然后操作数据，最后写入物理存储设备。

基于数据集操作的系统对两种应用的处理并不高效：一是迭代式的算法，这在图应用和机器学习领域很常见，二是交互式数据挖掘工具（反复查询一个数据子集）。这两种情况下，将数据保存在内存中能够极大地提高性能。基于数据集的方式不能够复用曾经的计算结果或中间计算结果，Hadoop 每次作业都从磁盘上读写数据，而且第二次作业运行时会再次从磁盘上读写数据，不能基于内存共享数据，这种数据集系统对通用的应用的操作处理并不高效，因为通用的处理一般都是用迭代的（如机器学习和图计算），另外假设要对处理的结果进行复用的话，例如，想要复用图 1-1 中 Job#1 的结果，这时候必须在磁盘上重读，不能基于内存复用，只能再次执行 Job#1，不会从内存中复用上次的结果。基于数据

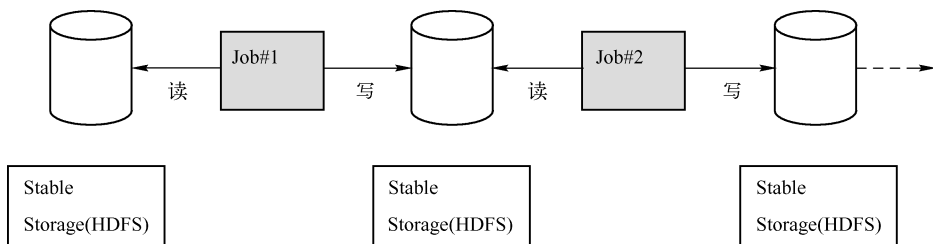


图 1-1　工作模型图

集的系统对交互式数据挖掘工具也不太适用。如果能将数据保存到内存中，其性能将会得到大大提升。

数据集的特性有位置感知、容错性和负载均衡等。在众多特性中，最难实现的是容错性。一般来说，分布式数据集的容错性有两种方式：数据检查点和记录数据更新。当面向大规模数据分析时，数据检查点操作成本很高：需要通过数据中心的网络链接在各台机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源（在内存中复制可以减少需要缓存的数据量，而存储到磁盘则会拖慢应用程序）。

这里所讲的 RDD 就是在这种背景下所产生的。Spark 的 RDD 是一种基于工作集的工作模式。无论数据集还是工作集，它们都有一些共同的特征，如位置感知、容错和负载均衡等，为了有效地实现容错，RDD 本身提供了一种高度受限的共享内存模型，它是只读的记录分区的集合，只能够通过从外界读取文件，或者说由其他的 RDD 来产生。RDD 的读操作可以精确到一条记录，RDD 的写操作则是批量的。RDD 的模型特别适合迭代，因为后面的 RDD 都是依据前面的 RDD 产生的，或者从外部读取数据而产生的，这就有一种前后的依赖关系，创建 RDD 的一系列转换被记录下来（即 Lineage 机制），以便恢复失去的数据。

Spark 的 RDD 为基于工作集的应用提供了更为通用的抽象，用户可以对中间结果进行显式的命名和物化，控制其分区，还能执行用户选择的特定操作（而不是在运行时去循环执行一系列 MapReduce 步骤），Spark 为工作集的应用提供了基本的抽象，同时又具有以 Hadoop 为代表的数据流模型的优势（如自动位置的感知、自动容错、伸缩性和良好的调度等），Spark 的 RDD 之间具有依赖关系且高度抽象，编程模型更容易，容错更好。一些算法，如逻辑回归、k 幂次数列等，都会在多个作业中重用或共用计算结果，在多个共享数据集中交互式查询。RDD 本身采用分布式内存计算的抽象容错机制解决了多步骤迭代，在一个共享数据集中执行多个交互式查询（多个 Job 中重用计算数据），这是 RDD 的使用场景。RDD 不太适合那些异步更新共享状态的应用，如并行 Web 爬虫。RDD 的目标是为大多数分析型应用提供有效的编程模型。

Section 1.2 RDD 的基本概念

1.2.1 RDD 的定义

RDD（Resilient Distributed Datasets，弹性分布式数据集）是分布式内存的一个抽象概念，是一种高度受限的共享内存模型，即 RDD 是只读的记录分区的集合，能横跨集群的所有结点进行并行计算，是一种基于工作集的应用抽象。

RDD 底层存储原理为：其数据分布存储于多台机器上，事实上，每个 RDD 的数据都以 Block 的形式存储在多台机器上。图 1-2 所示是 Spark 的 RDD 存储架构图，其中每个 Executor 会启动一个 BlockManagerSlave，并管理一部分 Block；而 Block 的元数据由 Driver 结点上的

BlockManagerMaster 保存，BlockManagerSlave 生成 Block 后向 BlockManagerMaster 注册该 Block，BlockManagerMaster 管理 RDD 与 Block 的关系，当 RDD 不再需要存储时，将向 BlockManagerSlave 发送指令删除相应的 Block。

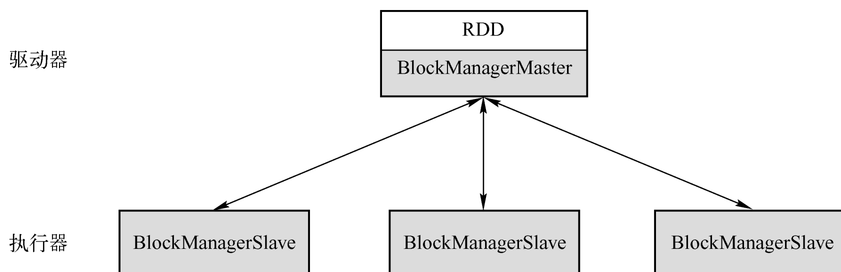


图 1-2　RDD 存储架构图

BlockManager 管理 RDD 的物理分区，每个 Block 就是结点上对应的一个数据块，可以存储在内存或者磁盘上。而 RDD 中的 Partition 是一个逻辑数据块，对应相应的物理块 Block。本质上，一个 RDD 在代码中相当于是数据的一个元数据结构，存储着数据分区及其逻辑结构的映射关系，存储着 RDD 之前的依赖转换关系。

在此需要对 BlockManager 进行简要的解释，BlockManager 的部分源代码如下。

```
/**
 * 在每个结点上运行管理 Block(Driver 和 Executors),它提供一个接口来检索本地和远程
   的存储变量
 * 如 memory、disk 和 off-heap
 * 请注意,在使用 BlockManager 之前必须要先 initialize()
 */
private[spark] class BlockManager(
    executorId:String,
    rpcEnv:RpcEnv,
    //负责对各个结点的 BlockManager 内部管理的数据的元数据进行管理与维护
    val master:BlockManagerMaster,
    defaultSerializer:Serializer,
    val conf:SparkConf,
    memoryManager:MemoryManager,
    mapOutputTracker:MapOutputTracker,
    shuffleManager:ShuffleManager,
    blockTransferService:BlockTransferService,
    securityManager:SecurityManager,
    numUsableCores:Int)
  extendsBlockDataManager with Logging
```

BlockManagerMaster 会持有整个 Application 的 Block 的位置、Block 所占用的存储空间等元数据信息，在 Spark 的 Driver 的 DAGScheduler 中就是通过这些信息来确认数据运行的本地

性的。Spark 支持重分区，数据通过 Spark 默认的或者用户自定义的分区器来决定数据块分布在哪些结点。RDD 的物理分区是由 Block – Manager 管理的，每个 Block 就是结点上对应的一个数据块，可以存储在内存或者磁盘中。而 RDD 中的 Partition 是一个逻辑数据块，对应相应的物理块 Block。本质上，一个 RDD 在代码中相当于是数据的一个元数据结构（一个 RDD 就是一组分区），存储着数据分区及 Block、Node 等的映射关系以及其他元数据信息，存储着 RDD 之前的依赖转换关系。分区是一个逻辑概念，Transformation 前后的新旧分区在物理上可能是同一块内存存储。

Spark 通过读取外部数据创建 RDD，或通过其他 RDD 执行确定的转换 Transformation 操作（如 map、union 和 groubByKey）而创建，从而构成了线性依赖关系或者说血统关系（lineage），在数据分片丢失时可以从依赖关系中恢复自己独立的数据分片，对其他数据分片或计算机没有影响，基本没有检查点开销，使得实现容错的开销很低，失效时只需要重新计算那些 RDD 分区，就可以在不同结点上并行执行，而不需要回滚（Roll Back）整个程序。关于落后任务（即运行很慢的结点）是通过任务备份，重新调用执行进行处理的。

因为 RDD 本身支持基于工作集的运用，所以可以使 Spark 的 RDD 持久化（Persist）到内存中，在并行计算中高效重用。在进行多个查询时，就可以显性地将工作集中的数据缓存到内存中，为后续查询提供复用，这极大地提升了查询的速度。在 Spark 中，一个 RDD 就是一个分布式对象集合，每个 RDD 可分为多个片（Partitions），而分片可以在集群环境的不同结点上计算。

RDD 作为泛型的抽象的数据结构（如下面源代码的第 8 行代码），支持两种计算操作算子：Transformation（变换）与 Action（行动）。并且 RDD 的写操作是粗粒度的，读操作既可以是粗粒度的也可以是细粒度的。

```
1.  /** 每个 RDD 都有 5 个主要特性 *
2.    *   – 分区列表
3.    *   – 每个分区都有一个计算函数
4.    *   – 依赖于其他 RDD 的列表
5.    *   – 数据类型(Key – Value)的 RDD 分区器
6.    *   – 每个分区都有一个分区位置列表
7.    */
8.  abstract class RDD[T:ClassTag](
9.        @transient private var _sc:SparkContext,
10.       @transient private vardeps:Seq[Dependency[_]]
11.     )extendsSerializable with Logging
```

在此需要对 SparkContext 做出解释：SparkContext 是 Spark 功能的主要入口点，一个 SparkContext 代表一个集群连接，可以用其在集群中创建 RDD、累加变量和广播变量等，在每一个可用的 JVM 中只有一个 SparkContext，在创建一个新的 SparkContext 之前必须先停止该 JVM 中可用的 SparkContext，这种限制可能最终会被修改。SparkContext 被实例化时需要一个 SparkConf 对象去描述应用的配置信息，在这个配置对象中设置的信息会覆盖系统默认的配置。

1.2.2 RDD 五大特性

1. 分区列表（a list of partitions）

Spark RDD 是被分区的，每一个分区都会被一个计算任务（Task）处理，分区数决定了并行计算的数量，RDD 的并行度默认从父 RDD 传给子 RDD。默认情况下，一个 HDFS 上的数据分片就是一个 partiton，RDD 分片数决定了并行计算的力度，可以在创建 RDD 时指定 RDD 分片个数，如果不指定分区数量，当 RDD 从集合创建时，则默认分区数量为该程序所分配到的资源的 CPU 核数（每个 Core 可以承载 2 ～ 4 个 partition），如果是从 HDFS 文件创建，默认为文件的 Block 数。

2. 每一个分区都有一个计算函数（a function for computing each split）

每个分区都会有计算函数，Spark 的 RDD 的计算函数是以分片为基本单位的，每个 RDD 都会实现 compute 函数，对具体的分片进行计算，RDD 中的分片是并行的，所以是分布式并行计算，有一点非常重要，就是由于 RDD 有前后依赖关系，遇到宽依赖关系，如 reduceByKey 等这些操作时划分成 Stage，Stage 内部的操作都是通过 Pipeline 进行的，在具体处理数据时它会通过 BlockManager 来获取相关的数据，因为具体的 split 要从外界读数据，也要把具体的计算结果写入外界，所以用了一个管理器，具体的 split 都会映射成 BlockManager 的 Block，而具体的 split 会被函数处理，函数处理的具体形式是以任务的形式进行的。

3. 依赖于其他 RDD 的列表（a list of dependencies on other RDDs）

由于 RDD 每次转换都会生成新的 RDD，所以 RDD 会形成类似流水线一样的前后依赖关系，当然宽依赖就不类似于流水线了，宽依赖后面的 RDD 具体的数据分片会依赖前面所有的 RDD 的所有数据分片，这个时候数据分片就不进行内存中的 Pipeline，一般都是跨机器的，因为有前后的依赖关系，所以当有分区的数据丢失时，Spark 会通过依赖关系进行重新计算，从而计算出丢失的数据，而不是对 RDD 所有的分区进行重新计算。RDD 之间的依赖有两种：窄依赖（Narrow Dependency）和宽依赖（Wide Dependency）。RDD 是 Spark 的核心数据结构，通过 RDD 的依赖关系形成调度关系。通过对 RDD 的操作形成整个 Spark 程序。

RDD 有窄依赖和宽依赖两种不同类型的依赖，其中的窄依赖指的是每一个 parent RDD 的 Partition 最多被 child RDD 的一个 Partition 所使用，而宽依赖指的是多个 child RDD 的 Partition 会依赖于同一个 parent RDD 的 Partition。可以从两个方面来理解 RDD 之间的依赖关系，一方面是 RDD 的 parent RDD 是什么，另一方面是依赖于 parent RDD 的哪些 Partion；根据依赖于 parent RDD 的哪些 Partion 的不同情况，Spark 将 Dependency 分为宽依赖和窄依赖两种。Spark 中的宽依赖指的是生成的 RDD 的每一个 partition 都依赖于父 RDD 所有的 partition，宽依赖典型的操作有 groupByKey、sortByKey 等，宽依赖意味着 Shuffle 操作，这是 Spark 划分 Stage 的边界的依据，Spark 中的宽依赖支持两种 Shuffle Manager，即 HashShuffleManager 和 SortShuffleManager，前者是基于 Hash 的 Shuffle 机制，后者是基于排序的 Shuffle 机制。

4. key – value 数据类型的 RDD 分区器（a Partitioner for key – value RDDS）、控制分区策略和分区数

每个 key – value 形式的 RDD 都有 Partitoner 属性，它决定了 RDD 如何分区。当然，Partiton 的个数还决定了每个 Stage 的 Task 个数。RDD 的分片函数可以分区（Partitioner），可传入相关的参数，如 HashPartitioner 和 RangePartitioner，它本身针对 key – value 的形式，如果不是 key – value 的形式它就不会有具体的 Partitioner，Partitioner 本身决定了下一步会产生多少并行的分片，同时它本身也决定了当前并行（Parallelize）Shuflle 输出的并行数据，从而使 Spark 具有能够控制数据在不同结点上分区的特性，用户可以自定义分区策略，如 Hash 分区等。Spark 提供了 partitionBy 运算符，能通过集群对 RDD 进行数据再分配来创建一个新的 RDD。

5. 每个分区都有一个优先位置列表（a list of preferred locations to compute each split on）

优先位置列表会存储每个 Partition 的优先位置，对于一个 HDFS 文件来说，就是每个 Partition 块的位置。观察运行 Spark 集群的控制台就会发现，Spark 在具体计算、具体分片以前，它已经清楚地知道任务发生在哪个结点上，也就是说任务本身是计算层面的、代码层面的，代码发生运算之前它就已经知道它要运算的数据在什么地方，有具体结点的信息。这就符合大数据中数据不动代码动的原则。数据不动代码动的最高境界是数据就在当前结点的内存中。这时候有可能是 Memory 级别或 Tachyon 级别的，Spark 本身在进行任务调度时会尽可能地将任务分配到处理数据的数据块所在的具体位置。据 Spark 的 RDD. Scala 源代码函数 getParferredLocations 可知，每次计算都符合完美的数据本地性。

可在 RDD 类源代码文件中找到 4 个方法和 1 个属性，对应上述所阐述的 RDD 的五大特性，源代码剪辑如下。

```
/** 返回一个 RDD 分区列表,这个方法仅被调用一次,它是安全地执行一次耗时计算 */
protected def getPartitions:Array[Partition]
/** 通过子类来实现给定分区的计算 */
  @DeveloperApi
def compute(split:Partition,context:TaskContext):Iterator[T]
/** 返回对父 RDD 的依赖列表,这个方法仅被调用一次,它是安全地执行一次耗时计算 */
protected defgetDependencies:Seq[Dependency[_]] = deps
/** 可选的,分区的方法,可指定如何分区 */
@transient val partitioner:Option[Partitioner] = None
/** 可选的,指定优先位置,输入参数是 split 分片,输出结果是一组优先的结点位置 */
protected def getPreferredLocations(split:Partition):Seq[String] = Nil
```

在此需要对 TaskContext、Partitioner 和 Partition 等概念做出解释，TaskContext 是读取或改变执行任务的环境，用 org. apache. spark. TaskContext. get( ) 可返回当前可用的 TaskContext，可以调用内部的函数访问正在运行任务的环境信息。Partitioner 是一个对象，定义了如何在 key – value 类型的 RDD 的元素中用 key 分区，从 0 到 numPartitions – 1 区间内映射每一个 key 到 partition ID。Partition 是在一个 RDD 的分区标识符，源代码如下。

```
trait Partition extendsSerializable {
  //在它的父 RDD 的分区索引
  def index:Int
  //最好默认实现 HashCode
  override defhashCode():Int = index
}
```



1.2.3 RDD 弹性特性的 7 个方面

RDD 作为弹性分布式数据集，它的弹性具体体现在以下 7 个方面。

1. 自动进行内存和磁盘数据存储的切换

Spark 会优先把数据放到内存中，如果内存实在放不下，则会放到磁盘里面，不但能计算内存放下的数据，也能计算内存放不下的数据。如果实际数据大于内存，则要考虑数据放置策略和优化算法。当应用程序内存不足时，Spark 应用程序将数据自动从内存存储切换到磁盘存储，以保障其高效运行。

2. 基于 Lineage（血统）的高效容错机制

Lineage 是基于 Spark RDD 的依赖关系来完成的（依赖分为窄依赖和宽依赖两种形态），这种前后的依赖关系可恢复失去的数据。每个操作只关联其父操作，各个分片的数据之间互不影响，出现错误时只需恢复单个 Split 的特定部分即可。常规容错有两种方式：一个是数据检查点，另一个是记录数据的更新。数据检查点的基本工作方式就是通过数据中心的网络链接不同的机器，然后每次操作时都要复制数据集，就相当于每次都有一个拷贝，拷贝是要通过网络的，网络带宽就是分布式的瓶颈，对存储资源也是很大的消耗。记录数据更新就是每次数据变化了就记录一下，这种方式不需要重新复制一份数据，但是比较复杂，会消耗性能。Spark 的 RDD 通过记录数据更新的方式很高效，原因为：一是 RDD 是不可变的且是 Lazy 级别；二是 RDD 的写操作是粗粒度的。但是 RDD 的读操作既可以是粗粒度的也可以是细粒度的。

3. Task 如果失败会自动进行特定次数的重试

默认重试次数为 4 次。源代码如下。

```
private[spark] class TaskSchedulerImpl(
    val sc:SparkContext,
    valmaxTaskFailures:Int,
    isLocal:Boolean = false)
  extends TaskScheduler with Logging //继承任务调度器、日志 trait
{
  def this(sc:SparkContext) = this(sc,sc.conf.getInt("spark.task.maxFailures",4))
  …
}
```

TaskSchedulerImpl 是底层的任务调度接口 TaskScheduler 的实现，这些 Schedulers 从每一个 Stage 中的 DAGScheduler 中获取 TaskSet，运行它们。DAGScheduler 是高层调度，它计算每个 Job 的 Stages 的 DAG，然后提交给 Stages，用 TaskSets 的形式启动底层 TaskScheduler 调度在集群中运行。

4. Stage 如果失败会自动进特定次数的重试

这样 Stage 对象可以跟踪多个 StageInfo（存储 SparkListeners 监听到的 Stage 信息，将 Stage 信息传递给 Listeners 或 web UI。默认重试次数为 4 次，且可以直接运行计算失败的阶段，只计算失败的数据分片，Stage 源代码如下。

```
private[scheduler] abstract class Stage(
      val id:Int,
      val rdd:RDD[_],
      val numTasks:Int,
      val parents:List[Stage],
      val firstJobId:Int,
      val callSite:CallSite)
   extends Logging {
   //Partition 的个数
   val numPartitions = rdd. partitions. length
   //属于这个工作集的 Stage
   val jobIds = new HashSet[Int]
   val pendingPartitions = new HashSet[Int]
   //用于此 Stage 的下一个新 Attempt 的标识 ID
   private var nextAttemptId:Int = 0
   val name:String = callSite. shortForm
   val details:String = callSite. longForm
   private var _internalAccumulators:Seq[Accumulator[Long]] = Seq. empty
   //Stage 内部所有任务共享的累加器
   def internalAccumulators:Seq[Accumulator[Long]] = _internalAccumulators
   /**
     * 重新初始化与该 Stage 相关联的内部累加器
     * 当属于这个 Stage 的任务的一个子集已经完成时,称为一次提交,否则
 * 重新初始化内部累加器,这里又将覆盖部分任务
 */
   def resetInternalAccumulators():Unit = {
      _internalAccumulators = InternalAccumulator. create(rdd. sparkContext)
   }
   /**
 * 最新的[StageInfo] object 指针。这需要在这里被初始化,
     * 任何 Attempts 都是被创造出来的,因为 DAGScheduler 使用 StageInfo
     * 告诉 SparkListeners 工作开始时(即发生之前的任何阶段已经创建)
```



```
      */
    private var _latestInfo:StageInfo = StageInfo. fromStage(this,nextAttemptId)
    /**
      * 设置 Stage Attempt IDs,当失败时可以读取失败信息,
      * 跟踪这些失败,为了避免无休止的重复不断失败
      * 在同一个 Stage 中多个 Tasks 任务尝试失败时,我们使用 Hashset 集合记录每一个
      * 尝试失败的 ID 号,这样可避免记录重复的失败情况(spark - 5945)
      */
  private val fetchFailedAttemptIds = new HashSet[Int]
    private[scheduler] def clearFailures():Unit = {
      fetchFailedAttemptIds. clear()
    }
    /**
      * 检查是否应该中止由于连续多次读取失败的 Stage
      * 如果失败的次数超过允许的次数,此方法更新失败 Stage Attempts 和返回的运行集
      */
  private[scheduler] def failedOnFetchAndShouldAbort(stageAttemptId:Int):Boolean = {
      fetchFailedAttemptIds. add(stageAttemptId)
      fetchFailedAttemptIds. size >= Stage. MAX_CONSECUTIVE_FETCH_FAILURES
    }
    /** 在 Stage 中创建一个新的 attempt  */
    def makeNewStageAttempt(
        numPartitionsToCompute:Int,
        taskLocalityPreferences:Seq[Seq[TaskLocation]] = Seq. empty):Unit = {
      _latestInfo = StageInfo. fromStage(
        this,nextAttemptId,Some(numPartitionsToCompute),taskLocalityPreferences)
      nextAttemptId + = 1
    }
    /** 返回当前 Stage 中最新的 StageInfo   */
    def latestInfo:StageInfo = _latestInfo
    override final defhashCode():Int = id
    override final def equals(other:Any):Boolean = other match {
      case stage:Stage =>  stage  != null && stage. id == id
      case _ =>  false
    }
    //返回需要重新计算的分区标识的序列
    def findMissingPartitions():Seq[Int]
  }
  private[scheduler] object Stage {
    // 允许在一个 Stage 的中止的连续故障数
  val MAX_CONSECUTIVE_FETCH_FAILURES = 4
  }
```

Stage 是 Spark Job 运行时具有相同逻辑功能且并行计算任务的一个基本单元。Stage 中所有的任务都依赖同样的 Shuffle，每个 DAG 任务通过 DAGScheduler 在 Stage 的边界处发生 Shuffle 形成 Stage，然后 DAGScheduler 运行这些阶段的拓扑顺序。每一个 Stage 都可能是 ShuffleMapStage，如果是 ShuffleMapStage 则跟踪每个输出结点（nodes）上的输出文件分区，它的任务结果是输入其他的 Stage，或者输入一个 ResultStage，若输入一个 ResultStage，这个 ResultStage 的任务直接在这个 RDD 上运行计算这个 Spark Action 的函数（如 count()、save()等），并生成 shuffleDep 等字段描述 Stage 和生成变量（如 outputLocs 和 numAvailableOutputs），为跟踪 map 输出做准备。每一个 Stage 都会有 firstjobid，确定第一个提交 Stage 的 Job，当使用 FIFO 调度时，会使得其前面的 Job 先行计算或快速恢复（失败时）。

在此需要对 ShuffleMapStage、ResultStage 和 SparkListener 等做出解释。ShuffleMapStage 是 DAG 产生数据进行 Shuffle 的中间阶段，它发生在每次 Shuffle 操作之前，可能包含多个 pipelined 操作，ResultStage 阶段捕获函数在 RDD 的分区上进行 Action 算子计算结果，有些 Stages 并不是运行在 RDD 的所有分区上，如first()、lookup()等。SparkListener 是 Spark 调度器的事件监听接口，注意这个接口随着 Spark 的版本的不同会跟随着变化。

5. Checkpoint 和 Persist（检查点，持久化），可主动或被动触发

Checkpoint 是对 RDD 进行的标记，会产生一系列的文件，且所有父依赖都会被 Remove，是整个依赖（Lineage）的终点。Checkpoint 也是 Lazy 级别的。persist 后 RDD 工作时，每一个工作结点都会把计算的分片结果保存在内存或磁盘中，下一次如果要对相同的 RDD 进行其他的 Action 计算，就可以重用。

因为用户只与 Driver Program 交互，因此只能用 RDD 中的 cache()方法去 cache 用户能看到的 RDD。所谓能看到，是指经过 Transformation 算子处理后生成的 RDD，而某些在 Transformation 算子中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，如 reduceByKey()中生成的 ShuffleRDD、MapPartitionsRDD 是不能被用户直接 cache 的。在 Driver Program 中设定 RDD. cache()后，系统如何对进行 cache？首先在计算 RDD 的 Partition 之前就去判断 Partition 要不要被 cache，如果要被 cache 的话，先将 Partition 计算出来，然后 cache 到内存。Cache 只使用 memory，写到磁盘的话就 checkpoint。调用 RDD. cache()后，RDD 就变成 persistRDD 了，其 StorageLevel 为 MEMORY_ONLY，persistRDD 会告知 Driver 说自己是需要被 persist 的。此时会调用 RDD. iterator()，RDD 的 iterator()的源代码如下。

```
/**
  * RDD 的内部方法,将从合适的缓存中读取,否则计算它
  * 这不应该被用户直接使用,但可用于实现自定义的子 RDD
  */
final def iterator(split:Partition,context:TaskContext):Iterator[T] = {
  if (storageLevel != StorageLevel.NONE){//存储级别不等于 NONE
    getOrCompute( split,context)
  } else {
    computeOrReadCheckpoint(split,context)
```

```
    }
}
```

当 RDD. iterator( ) 被调用时，也就是要计算该 RDD 中的某个 Partition 时，会先去 cacheManager 中获取一个 blockId，然后去 BlockManager 中匹配该 Partition 是否被 checkpoint，如果是就不用计算该 Partiton 了，直接从 checkpoint 中读取该 Partition 的所有 records 放入 ArrayBuffer 里面；如果没有被 checkpoint 过，则先将 Partiton 计算出来，然后将其所有的 records 放到 cache 中。总体来说当 RDD 会被重复使用（不能太大）时，RDD 需要 cache。Spark 自动监控每个结点缓存的使用情况，利用最近最少使用的原则来删除老旧的数据。如果想手动删除 RDD，可以使用 RDD. unpersist( ) 方法。

此外，可以利用不同的存储级别存储每一个被持久化的 RDD。例如，它允许持久化集合到磁盘上、将集合作为序列化的 Java 对象持久化到内存中、在结点间复制结合或者存储集合到 Tachyon 中。可以通过传递一个 StorageLevel 对象给 persist( ) 方法设置这些存储级别。cache( ) 方法使用默认的存储级别 – StorageLevel. MEMORY_ONLY。RDD 根据 useDisk、useMemory、useOffHeap、deserialized 和 replication 这 5 个参数的组合提供了常用的 12 种存储级别，完整的存储级别介绍如下。

```
val NONE = newStorageLevel(false,false,false,false)
val DISK_ONLY = newStorageLevel(true,false,false,false)
val DISK_ONLY_2 = newStorageLevel(true,false,false,false,2)
val MEMORY_ONLY = newStorageLevel(false,true,false,true)
val MEMORY_ONLY_2 = newStorageLevel(false,true,false,true,2)
val MEMORY_ONLY_SER = newStorageLevel(false,true,false,false)
val MEMORY_ONLY_SER_2 = newStorageLevel(false,true,false,false,2)
val MEMORY_AND_DISK = newStorageLevel(true,true,false,true)
val MEMORY_AND_DISK_2 = newStorageLevel(true,true,false,true,2)
val MEMORY_AND_DISK_SER = newStorageLevel(true,true,false,false)
val MEMORY_AND_DISK_SER_2 = newStorageLevel(true,true,false,false,2)
//堆外存储
val OFF_HEAP = newStorageLevel(false,false,true,false)
```

在此需要对 StorageLevel 做出相应的解释。StorageLevel 是控制存储 RDD 的标志，每个 StorageLevel 记录 RDD 是否使用 Memory，或使用 ExternalBlockStore 存储，如果 RDD 脱离了 Memory 或 ExternalBlockStore，是否 drop 掉 RDD，是否保留数据在内存中的序列化格式，以及是否复制多个结点的 RDD 分区。另外，org. apache. spark. storage. StorageLevel 是单实例（singleton）对象，包含了一些静态常量和常用的存储级别，且可用单实例对象工厂方法（StorageLevel（...））创建定制化的存储级别。

Spark 的多个存储级别意味着在内存利用率和 CPU 利用率间的不同权衡。推荐通过下面的过程选择一个合适的存储级别：①如果 RDD 适合默认的存储级别（MEMORY_ONLY），就选择默认的存储级别。因为这是 CPU 利用率最高的选项，会使 RDD 上的操作尽可能地快。②如果不适合用默认级别，选择 MEMORY_ONLY_SER。选择一个更快的序列化库来提



高对象的空间使用率，但是仍能够相当快地访问。③除非算子计算 RDD 花费较大或者需要过滤大量的数据，否则不要将 RDD 存储到磁盘上，否则重复计算一个分区就会和从磁盘上读取数据一样慢。④如果希望更快地恢复错误，可以利用 replicated 存储级别，所有的存储级别都可以通过 replicated 计算丢失的数据来支持完整的容错，另外 replicated 的数据能在 RDD 上继续运行任务，而无须重复计算丢失的数据。在拥有大量内存的环境中或者多应用程序的环境中，OFF_HEAP 将对象从堆中脱离出来序列化，然后存储在一大块内存中，这就像它存储在磁盘上一样，但它仍在 RAM 中。对象在这种状态下不能直接使用，它们必须首先反序列化，也不受垃圾收集机制影响。OFF_HEAP 具有如下优势：OFF_HEAP 运行多个执行者共享的 Tachyon 中相同的内存池；OPP_HEAP 显著地减少来自 GC 回收的划分；如果单个的 Executor 崩溃，缓存的数据不会丢失。

6. 数据调度弹性（DAGScheduler、TASKScheduler 和资源管理无关）

Spark 将执行模型抽象为通用的有向无环图计划（DAG），这可以将多 Stage 的任务串联或并行执行，从而无须将 Stage 的中间结果输出到 HDFS 中，当发生运行结点故障时可有其他可用结点代替该故障结点运行。

7. 数据分片的高度弹性（coalesce）

Spark 进行数据分片时，默认将数据放在内存中，如果内存放不下，一部分会放在磁盘上进行保存。coalesce 的源代码如下。

```
/**
  * 返回一个新的 RDD,恰好有 numpartitions 分区,类似的合并定义在 RDD 序列
  * 此操作的结果在一个窄依赖,如果是从 1000 个分区到 100 个分区,就
  * 不会有一个 Shuffle,而不是每 100 个新的分区将要求 10 个当前分区
  */
case class Coalesce(numPartitions:Int,child:SparkPlan)extends UnaryNode {
  override def output:Seq[Attribute] = child.output
  override def outputPartitioning:Partitioning = {
    if(numPartitions == 1)SinglePartition
    else UnknownPartitioning(numPartitions)
  }
  protected override def doExecute():RDD[InternalRow] = {
    child.execute().coalesce(numPartitions,shuffle = false)
  }
  override def canProcessUnsafeRows:Boolean = true
}
```

例如，在计算的过程中会产生很多的数据碎片，此时产生一个 Partition 可能会非常小，如果一个 Partition 非常小，每次都会消耗一个线程去处理的话，可能会降低它的处理效率，这时候需要考虑把许多小的 Partition 合并成一个较大的 Partition 去处理，这样会提升效率。另一方面，有可能内存不是那么多，而每个 Partition 的数据 Block 比较大，这时候需要考虑把 Partition 变成更小的数据分片，这样会让 Spark 处理更多的批次但是不会出现 OOM 异常（内存溢出）。

Section

1.3 创建 RDD 的方式

1.3.1 通过已经存在的 Scala 集合创建 RDD

Spark 程序首先做的第一件事就是创建一个 SparkContext 对象，它告诉 Spark 如何访问一个集群。Spark 的 Shell 给用户提供了一个简单强大的工具来分析数据交互，通过运行 Spark 目录下的脚本即可启动它，同时 Shell 会创建一个 SparkContext 的变量 sc，启动代码如下。

```
1.    /bin/spark - shell
```

Spark 提供了两种创建 RDD 的方式：加载外部数据集和在驱动程序中并行集合。每个 RDD 都有 Partitioner 属性，它决定了该 RDD 如何分区，当然 Partition 的个数还将决定每个 Stage 的 Task 个数。当前 Spark 需要应用设置 Stage 的并行 Task 个数（配置项为：spark. default. parallelism），在未设置的情况下，子 RDD 会根据父 RDD 的 Partition 决定，例如，map 操作的子 RDD 的 Partition 与父 Partition 完全一致，Union 操作的子 RDD 的 Partition 个数为父 Partition 个数值之和。Stage 的并行 Task 的数量很大程度上决定 Spark 程序的性能。

并行集合（Parallelized collections）是通过在一个已有的集合（ScalaSeq）上调用 SparkContext 提供的 parallelize 方法实现的。集合中的元素被复制到一个可并行操作的分布式数据集中。例如，下面是如何创建一个数字 1 ～ 5 的集合代码。

```
1.    val data = Array(1,2,3,4,5)
2.    val distData = sc. parallelize(data)
```

一旦创建完成，分布式集合（distData）可以被并行操作。例如，可能会调用 distData. reduce((a,b) => a + b)将这个数组中的元素相加。

并行集合有一个很重要的参数，即切片数（slices），表示一个数据集的切片的份数。Spark 会在集群的每一个分片上运行一个任务，集群中的每个 CPU 通常需要 2 ～ 4 个分区。通常 Spark 会自动基于集群状态设置分区数目。也可以通过 parallelize 的第二个参数手动设置（如 sc. paralielize(data,10))。注意：某些地方在代码中使用数据分片（分区）的同义词来维持向后兼容性。

1.3.2 通过 HDFS 和本地文件系统创建 RDD

从 HDFS 等外部存储作为输入数据源，数据按照 HDFS 中的数据分布策略进行数据分区，HDFS 中的一个 Block 对应 Spark 的一个分区。生产环境中最常用的 RDD 创建方式是从 HDFS 来创建 RDD，SparkContext 中的 textFile 函数从 HDFS 读取文件，输出变量 file。file 是一个 RDD，实际是 HadoopRDD 实例。案例代码如下。



```
1.  val file = spark. textFile( "hdfs://..." )
2.  //过滤包含字符 info 的信息的行
3.  val infos = file. filter( line => line. contains( “info” ) )
4.  //统计所有的信息行的数量
5.  val counts = infos. count( )
```

第 1 行代码从 Hadoop 文件系统（HDFS）中定义了一个 RDD（即一个文本行集合），第 2 行代码用 filter 函数过滤带 info 的行输出 infos（infos 也是一个 RDD），第 3 行代码返回 infos 的行数。整个转换的过程如图 1-3 所示。

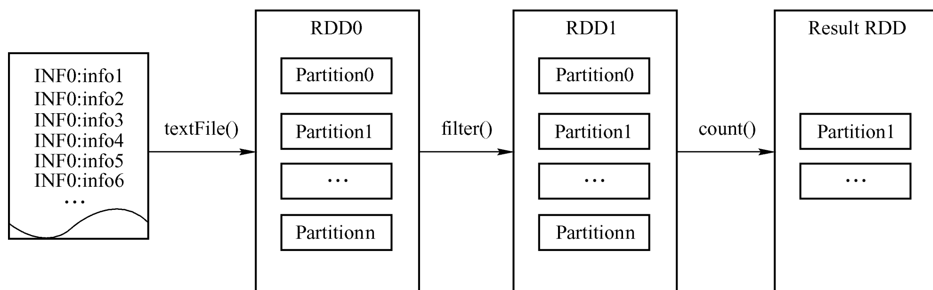


图 1-3　RDD 创建及操作示意图

本地文件系统创建 RDD，可测试大量数据的文件。下面用 Spark 源文件目录中的 RE-ADME. md 文件创建一个新的 RDD，代码如下。

```
scala > val textFile = sc. textFile( "README. md" )
textFile: spark. RDD[ String ] = spark. MappedRDD@ 2ee9b6e3
```

RDD 一旦创建完成，textFile 就可以做数据集操作。例如，可以用方法 map 和 Reduce 操作将所有行的长度相加：textFile. map(s => s. length). reduce((a, b) => a + b)。

1. 3. 3　其他的 RDD 的转换

从父 RDD 转换可得到新的 RDD。通过 Spark 内核给用户提供的 Transformation 来对 RDD 进行各种算子的转化，形成新的 RDD 实现算法。下面通过一段常用代码来讲解一下，代码如下。

```
scala > val linesWithSpark = textFile. filter( line => line. contains( "Spark" ) )
linesWithSpark: spark. RDD[ String ] = spark. FilteredRDD@ 7dd4af09
```

这里涉及两个 RDD，第一个 RDD 是 textFile，textFile 是一个 HadoopRDD 经过 map 后的 MapPartitionsRDD，第二个 RDD 是经过 filter 方法之后生成的一个 FilteredRDD，即返回一个新的 RDD（原 RDD 的子集）。

接下来将要讲解上述代码片段在 Spark 内部的处理流程。首先来看 textFile 方法，进入 SparkContext 这个类找到它，源代码如下。

```
/** 从 HDFS 读取一个文本文件,或从本地文件系统(所有结点上都可用),或任何 Hadoop
 * 支持的文件系统中,并返回一个字符串 RDD
 */
def textFile(
    path:String,
    //最小分区数,如果用户没有给定,默认是两个
    minPartitions:Int = defaultMinPartitions):RDD[String] = withScope {
    assertNotStopped()

    hadoopFile(path,classOf[TextInputFormat],classOf[LongWritable],
    classOf[Text],
    minPartitions).map(pair => pair._2.toString)
  }
```

由 SparkContext 类中 textFile 方法的源代码可知，hadoopFile 后面加了一个 map 方法，取 pair 的第二个参数，最后在 shell 里面可看到它是一个 MappredRDD。默认的 defaultMinPartitions 的大小为 2。源代码语句如下。

```
/**
 * Hadoop RDDs 中用户不指定分区数时的默认最小分区数
 * 注意所使用的 math.min 的 "defaultMinPartitions" 不能大于 2
 * 这个原因已在网站 https://github.com/mesos/spark/pull/718 中讨论过
 */
def defaultMinPartitions:Int = math.min(defaultParallelism,2)
```

查看 hadoopFile 方法的源代码可知，hadoopFile 做了 3 个操作，把 Hadoop 的配置文件保存到广播变量里，设置了路径，new 了一个 HadoopRDD 返回。HadoopFile 方法的源代码如下。

```
/**  得到一个 Hadoop 文件中任意 InputFormat 的 RDD */
    def hadoopFile[K,V](
    path:String,
    inputFormatClass:Class[_ <:InputFormat[K,V]],
    keyClass:Class[K],
    valueClass:Class[V],
    //最小分区数,如果用户没有给定,默认是两个
    minPartitions:Int = defaultMinPartitions):RDD[(K,V)] = withScope {
    assertNotStopped()
    //一个 Hadoop 的配置可以约 10 kb,这是相当大的,为了减少通信开销所以采用广播
      方式。
    val confBroadcast = broadcast(new SerializableConfiguration(hadoopConfiguration))
    val setInputPathsFunc = (jobConf:JobConf) => FileInputFormat.setInputPaths(jobConf,
  path)
  new HadoopRDD(
```

```
            this,
            confBroadcast,
            Some(setInputPathsFunc),
            inputFormatClass,
            keyClass,
            valueClass,
            minPartitions).setName(path)
    }
```

接下来分析 HadoopRDD，通过以下源代码可知，HadoopRDD 是一个对象。

```
private[spark] object HadoopRDD extends Logging
```

HadoopRDD 也是 RDD，由于 RDD 具有五大特性，所以重点关注它的 getPartitions 方法、compute 方法和 getPreferredLocations 方法。先看 getPartitions 方法，源代码如下。

```
override def getPartitions:Array[Partition] = {
      val jobConf = getJobConf()
      //SparkContext 初始化之前被调用,在此处添加 credentials
      SparkHadoopUtil.get.addCredentials(jobConf)
      //输入格式
      val inputFormat = getInputFormat(jobConf)
      //输入分片
      val inputSplits = inputFormat.getSplits(jobConf,minPartitions)
      //得到数组
      val array = new Array[Partition](inputSplits.size)
      for(i <- 0 until inputSplits.size){
         array(i) = new HadoopPartition(id,i,inputSplits(i))
      }
      array
   }
```

它调用的是 inputFormat 自带的 getSplits 方法来计算分片，然后把分片 HadoopPartition 包装到 Array 数组里面返回。

接下来分析 HadoopRDD 的 compute 方法，源代码如下。

```
override def compute(theSplit:Partition,context:TaskContext):InterruptibleIterator[(K,V)] = {
val iter = new NextIterator[(K,V)] {
      //转换成 HadoopPartition
      val split = theSplit.asInstanceOf[HadoopPartition]
      logInfo("Input split:" + split.inputSplit)
      val jobConf = getJobConf()
      val inputMetrics = context.taskMetrics.getInputMetricsForReadMethod(DataReadMethod.Hadoop)
```



```
    //为文件名设置线程局部变量
    split.inputSplit.value match {
      case fs:FileSplit => SqlNewHadoopRDDState.setInputFileName(fs.getPath.toString)
      case _ => SqlNewHadoopRDDState.unsetInputFileName()
    }
    /** 找到一个函数,该函数将返回这个线程读取的文件系统字节。做这之前先创建
    RecordReader,因为 RecordReader 的构造函数可以读取一些字节 */
    val bytesReadCallback = inputMetrics.bytesReadCallback.orElse {
      split.inputSplit.value match {
        case _:FileSplit | _:CombineFileSplit =>
          SparkHadoopUtil.get.getFSBytesReadOnThreadCallback()
        case _ => None
      }
    }
    inputMetrics.setBytesReadCallback(bytesReadCallback)
    //通过 Inputform 的 getRecordReader 来创建 InputSpit 的 Reader
    var reader:RecordReader[K,V] = null
    val inputFormat = getInputFormat(jobConf)
    HadoopRDD.addLocalConfiguration(new SimpleDateFormat("yyyyMMddHHmm").format
    (createTime),
        context.stageId,theSplit.index,context.attemptNumber,jobConf)
    reader = inputFormat.getRecordReader(split.inputSplit.value,jobConf,Reporter.NULL)
    // 注册一个任务完成回调,以关闭输入流
    context.addTaskCompletionListener{context => closeIfNeeded()}
  //调用 Reader 的 next 方法
    val key:K = reader.createKey()
    val value:V = reader.createValue()
    override def getNext():(K,V) = {
      try {
        finished = !reader.next(key,value)
      } catch {
        case eof:EOFException =>
          finished = true
      }
      if(!finished){
        inputMetrics.incRecordsRead(1)
      }
      (key,value)
    }
    override def close(){
      if(reader != null){
        SqlNewHadoopRDDState.unsetInputFileName()
```

```
            //关闭 reader 并释放它
            try {
              reader.close()
            } catch {
              case e:Exception =>
                if (!ShutdownHookManager.inShutdown()) {
                  logWarning("Exception in RecordReader.close()",e)
                }
            } finally {
              reader = null
            }
            if (bytesReadCallback.isDefined) {
              inputMetrics.updateBytesRead()
            } else if (split.inputSplit.value.isInstanceOf[FileSplit] ||
                    split.inputSplit.value.isInstanceOf[CombineFileSplit]) {
            try {
                //如果不能从 FS 中读取数据字节,分区的大小可能是不准确的
                inputMetrics.incBytesRead(split.inputSplit.value.getLength)
              } catch {
                case e:java.io.IOException =>
                  logWarning("Unable to get input size to set InputMetrics for task",e)
                }
              }
            }
          }
        }
        new InterruptibleIterator[(K,V)](context,iter)
    }
```

HadoopRDD 的输入值是一个 Partition，返回值是一个 Iterator[(K,V)]类型的数据，这里只关注两点即可：把 Partition 转换成 HadoopPartition，然后通过 InputSplit 创建一个 RecordReader；重写 Iterator 的 getNext 方法，通过创建的 reader 调用 next 方法读取下一个值。从这里可以看出，compute 方法是通过分片来获得 Iterator 接口的，遍历分片的数据。

接下来分析 getPreferredLocations 方法，该方法通过调用 InputSplit 接口的 getLocations 方法获得 RDD 所在的位置。源代码如下。

```
override def getPreferredLocations(split:Partition):Seq[String] = {
    val hsplit = split.asInstanceOf[HadoopPartition].inputSplit.value
    val locs:Option[Seq[String]] = HadoopRDD.SPLIT_INFO_REFLECTIONS match {
      case Some(c) =>
        try {
          vallsplit = c.inputSplitWithLocationInfo.cast(hsplit)
```

```
                //获得 RDD 的位置信息
                val infos = c.getLocationInfo.invoke(lsplit).asInstanceOf[Array[AnyRef]]
                Some(HadoopRDD.convertSplitLocationInfo(infos))
            } catch {
        case e:Exception =>
                //debug 异常日志信息
                  logDebug("Failed to use InputSplitWithLocations.",e)
                  None
            }
          case None => None
        }
        locs.getOrElse(hsplit.getLocations.filter(_ != "localhost"))
    }
```

最后分析 MapPartitionsRDD，首先了解一下 RDD 类里面的 map 方法，map 方法的源代码如下。

```
/** RDD 中的每个元素都经过函数 f 处理后返回一个新的 RDD */
def map[U:ClassTag](f:T => U):RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U,T](this,(context,pid,iter) => iter.map(cleanF))
}
```

从 map 方法的源代码可知，map 方法内部直接 new 了一个 MapPartitionsRDD，还把匿名函数 f 处理来再传进去，接下来展开 MapPartitionsRDD，MapPartitionsRDD 的源代码如下：

```
//RDD 对父 RDD 的每个分区提供的功能
private[spark] class MapPartitionsRDD[U:ClassTag,T:ClassTag](
    prev:RDD[T],
    //(TaskContext,分区索引,迭代器)
    f:(TaskContext,Int,Iterator[T]) => Iterator[U],
    preservesPartitioning:Boolean = false)
extends RDD[U](prev) {
override val partitioner = if(preservesPartitioning) firstParent[T].partitioner else
None
override def getPartitions:Array[Partition] = firstParent[T].partitions
override def compute(split:Partition,context:TaskContext):Iterator[U] =
  f(context,split.index,firstParent[T].iterator(split,context))
}
```

由 MapPartitionsRDD 的源代码可知，MapPartitionsRDD 把父类 RDD 的 partitioner、getPartitions 和 compute 给重写了，而且每个分区的 compute 中都用到了 firstParent[T]方法，firstParent[T]方法的源代码如下。

```
/** 返回第一个父 RDD */
  protected[spark] deffirstParent[U:ClassTag]:RDD[U] = {
    dependencies.head.rdd.asInstanceOf[RDD[U]]
  }
```

继续追踪方法 dependencies，dependencies 的源代码如下。

```
/** 得到这个 RDD 的依赖列表,要考虑这个 RDD 是否需要检查点 */
    final def dependencies:Seq[Dependency[_]] = {
    checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
      if (dependencies_ == null) {
        dependencies_ = getDependencies
      }
      dependencies_
    }
  }
```

由此可以得出两个结论：①getPartitions 直接沿用了父 RDD 的分片信息。②compute 函数是通过 RDD 遍历每一个数时接受一个匿名函数 f 进行处理的。现在应该可以看出 compute 函数有两个显著的作用，一个是在没有依赖的条件下，根据分片的信息生成遍历数据的 Iterable 接口；另一个是在有前置依赖的条件下，在父 RDD 的 Iterable 接口上遍历每个元素时再接受一个方法处理。

1.3.4 其他的 RDD 的创建

基于 DB 创建 RDD、基于 S3（S3 是一个公开的、云存储服务，Web 应用程序开发人员可以使用它存储数字资产，包括图片、视频、音乐和文档）创建 RDD，或基于数据流创建 RDD 等是比较常见的创建初始化 RDD 的方式。接下来的一个创建 RDD 的案例，是从 MySQL 关系型数据库中读取数据，最终得到 RDD 类型的数据。在 Spark 中提供了一个 JdbcRDD 的类，这个 JdbcRDD 执行一个 SQL 查询和 JDBC 链接读取结果，也就是说该 RDD 就是读取 JDBC 中的数据并转换成 RDD。

首先来看看该类的构造函数，源代码如下。

```
/**
 * @param getConnection //返回一个打开链接的函数
   这个 RDD 同样关注链接的关闭
 * @param sql sql 的查询语句
 *   查询必须包含两个用于分区结果参数占位符
 * @param lowerBound 最小值的第一个占位符
 * @param upperBound 最大值的第二个占位符
 *   上线和下线的边界
 * @param numPartitions 分区数
 *  给定一个下边界为 1,上边界为 20,分区数为 2
```

```
*   该查询被执行两次,一次(1,10)和一次(11,20)
* @param mapRow 一个结果集的函数,所需结果类型的一行
*是转换函数,将返回的 ResultSet 转成 RDD 需用的单行数据,此处可以选择 Array 或
*   其他,也可以是自定义的 case class。默认将 ResultSet 转换成一个 Object 数组
*/
class JdbcRDD[T:ClassTag](
sc: SparkContext,
getConnection:() => Connection,
sql: String,
lowerBound: Long,
upperBound: Long,
numPartitions: Int,
mapRow: (ResultSet) => T = JdbcRDD. resultSetToObjectArray _)
extends RDD[T](sc,Nil) with Logging
```

这个类带了很多参数，上述源代码处已给出解释。接下来是一段从 MySQL 中读取数据创建 RDD 的代码示例，代码如下。

```
def main(args:Array[String]){
val sc = new SparkContext("local","mysql")
val rddResult = new JdbcRDD(
sc,
() => {
Class. forName("com. mysql. jdbc. Driver"). newInstance()
DriverManager. getConnection("jdbc:mysql://localhost:3306/db","root","123456")
},
"SELECT content FROM test WHERE ID >= ? AND ID < = ?",
1,100,3,
  r => r. getString(1)). cache()
  }
```

上述代码用于读取 MySQL 中 test 表中的数据，并统计 ID 大于等于 1 且小于等于 100 的记录集合，得到结果集变量 rddResult，从代码中可以看出，JdbcRDD 的 sql 参数带有两个?的占位符，而这两个占位符是参数 lowerBound 和参数 upperBound 定义 where 语句的上下边界，从 JdbcRDD 类的构造函数可知，参数 lowerBound 和参数 upperBound 都只能是 Long 类型的，并不支持其他类型的比较，这个使得 JdbcRDD 使用的函数将比较有限。结合上述的两段代码可知，Spark 的 JdbcRDD 最终链接 MySQL 数据库并读取数据，并最终生成 RDD。

Section 1.4 RDD 算子

Spark 在运行中通过算子对 RDD 进行计算，算子是 RDD 中定义的函数，可以对 RDD 中

的数据进行转换和操作，如图 1-4 所示。

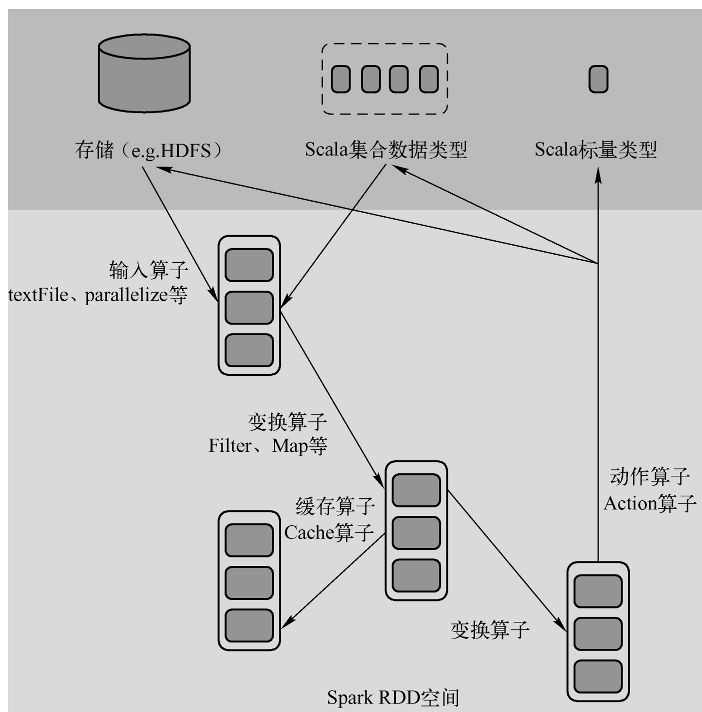


图 1-4　RDD 中的算子运算流程图

1. 输入

在 Spark 程序运行中，数据从外部数据空间（如分布式存储：textFile 读取 HDFS 等，parallelize 方法输入 Scala 集合或数据）输入 Spark，数据进入 Spark 运行时数据空间，转换为 Spark 中的数据块，通过 BlockManager 进行管理。

2. 运行

在 Spark 数据输入形成 RDD 后，可以通过变换算子（Transformation 算子），如 filter 等，对数据进行操作，并将 RDD 转化为新的 RDD，通过 Action 算子，触发 Spark 提交作业。如果数据需要复用，可以通过 Cache 算子将数据缓存到内存中。

3. 输出

程序运行结束后数据会输出 Spark 运行时的空间，存储到分布式存储中（如 saveAsTextFile 输出到 HDFS），或 Scala 类型的数据或集合中（collect 输出到 Scala 集合，count 返回 Scala int 型数据）。

Spark 中生成的不同 RDD 中，有的和用户逻辑显示的对应，如 map 操作生成 MapPartitionsRDD，而有的 RDD 则是 Spark 框架帮助用户隐式生成的，如 reduceByKey 操作时的 ShuffleRDD 等。Spark 的核心数据模型是 RDD，但 RDD 是一个抽象类，具体由各子类实现，如 MappedRDD、ShuffledRDD 等子类。在 Spark 的 reduceByKey 操作时会触发 Shuffle 的过程，在 Shuffle 之前，会有本地的聚合过程产生 MapPartitionsRDD，接着具体 Shuffle 会产生 ShuffledRDD，之后做全局的聚合生成结果 MapPartitionsRDD。Spark 将常用的大数据操作都转化

成 RDD 的子类，如图 1-5 所示。

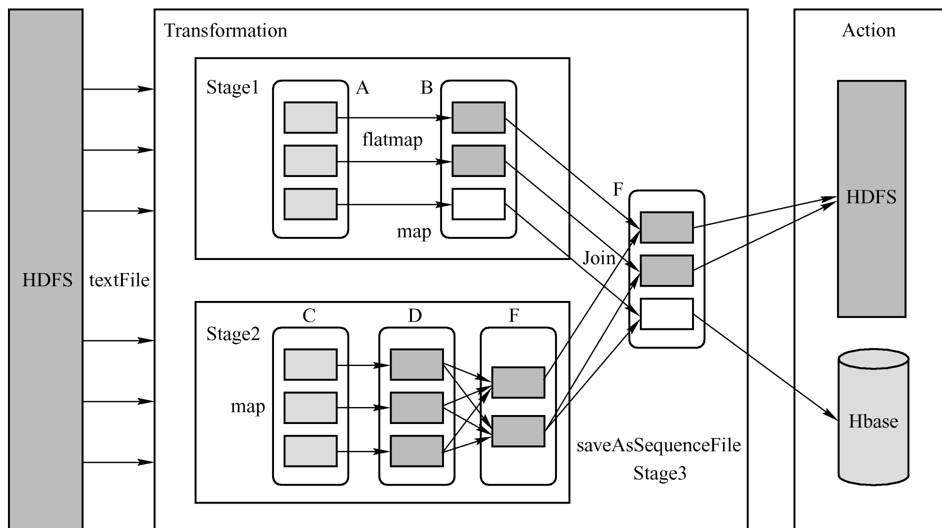


图 1-5　Spark 操作数据模型图

SparkRDD 中的算子大致可分为下列三大类。

1）Value 数据类型的 Transformation 算子，这种变换并不触发提交作业，针对处理的数据项是 Value 型的数据。

2）Key - Value 数据类型的 Transfromation 算子，这种变换并不触发提交作业，针对处理的数据项是 Key - Value 型的数据对。

3）Action 算子，这类算子会触发 SparkContext 提交 Job 作业，一个 Job 包含多个 Stage（至少一个 Stage），Stage 内部由一组完全相同的 Task 构成，这些 Task 只是处理的数据不同；一个 Stage 的开始就是从外部存储或者 Shuffle 结果中读取数据，一个 Stage 的结束是由于要发送 Shuffle 或者生成最终的计算结果。

对于 Spark 中的 Join 操作，如果每个 Partition 仅仅和特定的 Partition 进行 Join，那么就是窄依赖；对于需要 parent RDD 所有的 partition 进行 Join 的操作，即需要 Shuffle，此时就是宽依赖。RDD 的 saveAsTextFile 方法会首先生成一个 MapPartitionsRDD，该 RDD 通过调用 PairRDDFunctions 的 saveAsHadoopDataset 方法向 HDFS 等输出 RDD 数据的内容，并在最后调用 SparkContext 的 runJob 来真正向 Spark 集群提交计算任务。

默认情况下，每一个 Transformaton 过的 RDD 会在每次 Action 时重新计算一次，然而，可以使用 Persist（或 Cache）持久化一个 RDD 到内存中，可进行复用。根据 Action 算子的输出空间将 Action 算子进行分类：无输出、HDFS、Scala 集合和数据类型，RDD 模型适合粗粒度的全局数据并行计算，不支持细粒度的异步更新操作和增量迭代计算。

基于 RDD 的整个计算过程都是发生在 Worker 中的 Executor 中的。RDD 支持 3 种类型的操作：Transformation、Action，以及以 Persist 和 CheckPoint 为代表的控制类型的操作。RDD 一般会从外部数据源读取数据，经过多次 RDD 的 Transformation（中间为了容错和提高效率，有可能使用 Persist 和 CheckPoint），最终通过 Action 类型的操作一般会把结果写回外部存储系统。Spark Checkpoint 通过将 RDD 写入磁盘做检查点，是 Spark Lineage 容错机制的辅

助，Lineage 过长会造成容错成本过高，此时在中间阶段做检查点容错，如果之后有结点出现问题而丢失分区，从检查点的 RDD 开始重做 Lineage 就会减少开销。Checkpoint 主要适用于以下两种情况：①DAG 中的 Lineage 过长，若重算开销会太大，如在 PageRank、ALS 等；②尤其适合于在宽依赖上做 Checkpoint，这个时候就可以避免为 Lineage 重新计算而带来的冗余计算。

Section 1.5 RDD 的 Transformation 算子

1.5.1 Transformation 的定义

Transformation 是一种算法的描述。标记着需要进行操作的数据，但不真正执行。Transformation 具有 Lazy 特性，操作是延迟计算的。也就是说从一个 RDD 转换成另一个 RDD 的转换操作不是马上执行，需要等到有 Actions 操作或 Checkpoint 操作时，才真正触发操作。通过 Lazy 特性底层的 Spark 程序才能对应用程序进行优化。为什么通过 Lazy 特性 Spark 就能对应用程序进行优化呢？因为一直是延迟执行的 Spark 框架能够观察到很多运算步骤，观察到的越多，对其进行优化的机会就越多。

1.5.2 Transformation 在 RDD 中的角色定位及功能

Transformation 作为算子（算子是 RDD 中定义的函数，可以对 RDD 中的数据进行转换和操作）的一个抽象概念，从已经存在的数据集中创建一个新的数据集，Transformation 都具有 Lazy 特性，不立即计算 RDD 的结果，仅仅记录转换操作应用到哪些 RDD 上，Transformation 仅仅在执行 Action 时才进行计算（起作用），在 Action 之前不发生动作。

Transformation（变换）算子中可以将数据类型维度细分为 Value 数据类型和 Key－Value 对数据类型的 Transformation 算子。Value 型数据的算子封装在 RDD 类中可以直接使用，Key－Value 对数据类型的算子封装在 PairRDDFunctions 类中。

1.5.3 Transformation 操作的 Lazy 特性

Action 要数据的时候 Transformation 才会开始工作，Spark 里任何一个正常的作业都是没有进行计算的，最后一步要生成结果时，从后往前回溯父 RDD 有没有计算，以及其父 RDD 的父 RDD 有没有计算，从后往前推，这就是 Transformation 的 Lazy 特性，这样一个特性有一个非常大的好处，就是不需要结果时就不让这个作业进行计算，而在需要结果时它才发生具体的计算，这样就可以避免产生很多不必要的中间临时数据，这比较符合分布式并行计算的需求；另外一个层面是调度层面，最后一步要计算时，可以看到前面的所有步骤，看见的步骤越多，进行优化的机会就越多，所以 Spark 是基于 Lazy 特性进行操作、基于 Linage（血统

继承关系）来构建其整个调度系统的，最终形成了有向无环图（Directed Acyclic Graph, DAG）。需要特别注意的是，计算是在 Actions 或 Checkpoint 阶段，计算是从后往前回溯，优化由此开始（回溯和优化都发生在 Driver 上）。

1.5.4 通过实践说明 Transformation 的 Lazy 特性

下面以计算出 Spark 源文件目录中的 README. md 文件中找到最大的行数为例来说明 transformation 算子 Lazy 的特性，代码如下（其中的 map 算子即为 Transformation 算子）。

```
scala > importjava. lang. Math
importjava. lang. Math
scala > textFile. map(line => line. split(" "). size). reduce((a,b) => Math. max(a,b))
res5:Int = 15
```

map 创建一个新的 RDD，reduce 在 RDD 上调用 Math. max( ) 方法找到最大的行数，map 由于 Lazy 特性不立即计算，reduce 用于执行，是一个 action。此时 Spark 把计算分成多个 Task，并且运行在多个机器上，每台机器都运行自己的 map 和 reduce，最后只把结果返回给 Driver。要想证明上述代码的 map 算子具有的 Lazy 特性，只需在此基础上指定一个错误（不存在的）的文件路径，然后再执行即可，执行后并没有报错，所以上述代码的 map 算子是 Lazy 级别的，因为它并没有发生计算。

Section 1.6 RDD 的 Action 算子

1.6.1 Action 的定义

Action 是一种算法的描述，它通过 SparkContext 触发一个 Job 向用户程序返回一个值或一个结果。当要求返回 RDD（数据集）给 Driver 时，Transformations 会得到一个新的 RDD，方式很多，比如从数据源生成一个新的 RDD，从 RDD 生成一个新的 RDD。

1.6.2 Action 在 RDD 中的角色定位及功能

Action 算子会触发 SparkContext 的 runJob 方法提交作业（Job），触发 RDD DAG 的执行并将数据输出到 Spark 系统。Action 在 RDD（数据集）上进行计算之后返回一个值到 Driver，这样设计能让 Spark 运行得更高效。

Spark 中的 RDD 转换（Transformation）和动作（Action），每个操作都给出标识，其中方括号表示类型参数。前面说过 Transformation 是延迟操作，用于定义新的 RDD；而 Action 启动计算操作，并向应用程序返回值或向外部存储写数据。

在 Spark 中，RDD 被表示为对象，通过这些对象上的方法或函数调用 Transformation。定

义 RDD 之后，可以在 Action 中使用 RDD。Action 是向应用程序返回值或向存储系统导出数据的那些操作，如 count（返回 RDD 中的元素个数）、collect（返回元素本身）和 save（将 RDD 输出到存储系统）。在 Spark 中，只有在 Action 第一次使用 RDD 时，才会计算 RDD（即延迟计算）。这样在构建 RDD 时，运行时通过 Pipline 方法传输多个 Transformation 算子。另外 RDD 还允许根据关键字（key）指定分区顺序，这是一个可选的功能。目前支持 Hash 分区和 Range 分区，例如，应用程序请求将两个 RDD 按照同样的 Hash 方式进行分区（将同一机器上具有相同关键字的记录放在一个分区），以加速它们之间的 join 操作，多次迭代之间采用一致的分区置换策略进行优化，允许指定这样的优化。

RDD 提供了很多 Transformations 操作算子，每个转换操作算子都会生成新的 RDD，新的 RDD 依赖于原有的 RDD，这种 RDD 之间的依赖关系最终形成 DAG。DAG 经过 Shuffle 处理后形成 Stage。表 1-1 列出了一些常用的算子。

表 1-1　常用算子表

算 子 分 类	集体算子名称
Transformation	map(func)
	filter(func)
	flatMap(func)
	mapPartitions(func)
	sample(withReplacement,fraction,seed)
	union(withReplacement,fraction,seed)
	intersection(otherDataset)
	distinct([numTask])
	groupByKey([numTasks])
	reduceByKey(func,[numTasks])
	aggregateByKey(zeroValue)(seqOp,combOp,[numTasks])
	sortByKey([ascending],[numTasks])
	join(otherDataset,[numTasks])
Action	reduce(func)
	collect()
	count()
	first()
	take(n)

有些 hash 值针对 Key - Value 键值对可用，例如 groupByKey，另外，函数名与 Scala 及其他函数式语言中的 API 匹配，例如，map 是一对一的映射，而 flatMap 是将每个输入映射为一个或多个输出。用户可以通过 Partitioner 类获取 RDD 的分区顺序，然后将另一个 RDD 按照同样的方式分区。有些操作会自动产生一个 hash 或范围分区的 RDD，如 groupByKey、reduceByKey 和 sort 等。



Section
1.7 小结

本章主要介绍了以下内容：RDD 产生的技术背景及功能，RDD 基本概念，创建 RDD 的方式，RDD 的 Transformation 算子，RDD 的 Action 算子及 RDD 的 Transformation 及 Action 常用算子列表。RDD 是 Spark 大数据计算的基石，通过本章对 Spark RDD 内容的学习，可以进一步学习 Spark SQL、Spark Streaming、Spark ML 及图计算的学习打下坚实的基础。

第 2 章 RDD 的运行机制

本节内容主要讲解关于 RDD 运行机制。内容包括 RDD 依赖关系的意义与形成原因，基于 RDD 依赖关系形成有向无环图的过程，RDD 计算究竟是通过何种流程进行的，以及在计算中如何通过 RDD 缓存（Cache）、检查点（Checkpoint）等容错技术进行容错。

Section 2.1 RDD 依赖关系

RDD 的依赖关系是在整个 Spark 计算过程中尤为重要的一部分，涉及基于 RDD 计算的方方面面，还对于 DAG 的形成起着决定性的作用。本节主要从 RDD 依赖关系的分类角度分别讲解 RDD 的窄依赖和宽依赖。

2.1.1 窄依赖（Narrow Dependency）

RDD 的窄依赖是 RDD 中最常见的依赖关系，用于表示每一个父 RDD 中的 Partition 最多被子 RDD 的 1 个 Partition 所使用，如图 2-1 所示，父 RDD 有 2 ～ 3 个 Partition，每一个分区都只对应子 RDD 的 1 个 Partition。

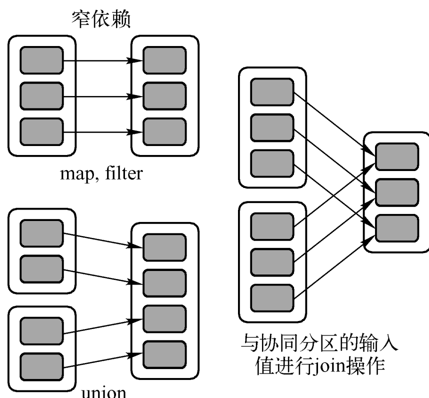


图 2-1 窄依赖关系图

在 Spark 的源代码中，把窄依赖分为两类：一类是一对一的依赖关系，在 Spark 中用 OneToOneDependency 来表示，它表示父 RDD 与子 RDD 的依赖关系是一对一的依赖关系，如图 2-1 中所示的 map、filter 和 join with inputs co - partitioned；第二类是范围依赖关系，在

Spark 中用 RangeDependency 表示，它表示父 RDD 与子 RDD 的依赖关系是一对一的范围内的依赖关系。如图 2-1 中所示的 union。

接下来通过 Spark 关于这两种依赖关系的源代码来分析这两种依赖关系具体在 Spark 中的实现方式。

Spark 关于依赖关系的源代码位于 Dependency. scala 文件中，其中关于 OneToOneDependency 的内容如下所示。

```
class OneToOneDependency[T](rdd:RDD[T])extends NarrowDependency[T](rdd){
override def getParents(partitionId:Int):List[Int] = List(partitionId)
```

可以看到，Spark 的重写方法引入了参数 partitionId，而在具体的方法中也使用了这个参数，这表明子 RDD 在通过 getParents 方法时，查询的是相同 partitionId 的内容，也就是说子 RDD 仅仅依赖父 RDD 中相同 partitionID 的 Partition。

而 Spark 中关于窄依赖还有第二种依赖关系，即 RangeDependency，Spark 源代码中关于它的内容如下所示。

```
override def getParents(partitionId:Int):List[Int] = {
if (partitionId >= outStart && partitionId < outStart + length){
List(partitionId - outStart + inStart)        }
else {          Nil       }
}
```

从代码中可以看到，RangeDependency 和 OneToOneDependency 最大的区别是其实现方法中出现了 outStart、length 和 instart 参数，子 RDD 在通过 getParents 方法查询对应的 Partition 时，会根据这个 partitionId 减去插入时的开始 ID，再加上它在父 RDD 中的位置 ID，换而言之，就是将父 RDD 中的 Partition 根据 partitionId 的顺序依次插入到子 RDD 中。

分析完 Spark 中的源代码，下边通过两个实例来让大家了解 RDD 窄依赖输出的结果。

对于 OneToOneDependency，采用 map 操作进行实验，实验代码和结果如下。

```
def main(args:Array[String]){
val num1 = Array(100,80,70)
val rddnum1 = sc.parallelize(num1)
val mapRdd = rddnum1.map(_ * 2)
mapRdd.collect().foreach(println)
}
```

结果为：200 160 140。

对于 RangeDependency，采用 union 操作进行实验，实验代码和结果如下。

```
def main(args:Array[String]){
  //创建数组 1
val data1 = Array("spark","scala","hadoop")
```

```
    //创建数组 2
val data2 = Array("SPARK","SCALA","HADOOP")
    //将数组 1 的数据形成 RDD1
val rdd1 = sc.parallelize(data1)
    //将数组 2 的数据形成 RDD2
val rdd2 = sc.parallelize(data2)
    //把 RDD1 与 RDD2 union
val unionRdd = rdd1.union(rdd2)
    //将结果收集并输出
unionRdd.collect().foreach(println)
}
```

结果为：spark scala hadoop SPARK SCALA HADOOP。

2.1.2 宽依赖（Shuffle Dependency）

RDD 的宽依赖是一种会导致计算时产生 Shuffle 操作的 RDD 操作，用来表示一个父 RDD 的 Partition 都会被多个子 RDD 的 Partition 所使用，如图 2-2 中的 groupByKey 所示，父 RDD 有 3 个 Partition，每个 Partition 中的数据会被子 RDD 中的两个 Partition 使用。

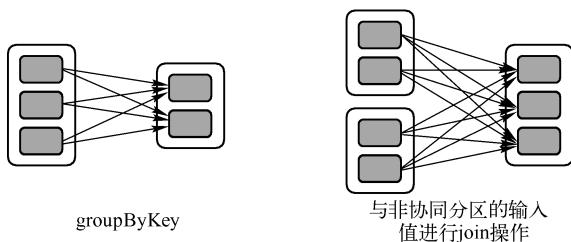


图 2-2　宽依赖关系图

有关宽依赖的源代码位于 Dependency.scala 文件的 Shuffle Dependency 方法，第 3 行产生了新的 ShuffleId，表明了宽依赖过程需要涉及 Shuffle 操作；途中后续的代码表示宽依赖进行时的 Shuffle 操作需要向 shuffleManager 注册信息。

```
1.  class ShuffleDependency[K:ClassTag,V:ClassTag,C:ClassTag]()
2.      //创建新的 ShuffleId
3.      val shuffleId:Int = _rdd.context.newShuffleId()
4.      //向 shuffleManagerage 注册 Shuffle
5.      val shuffleHandle:ShuffleHandle = _rdd.context.env.shuffleManager.registerShuffle(shuffleId,_
        rdd.partitions.size,this)
6.      _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
```

Spark 中，宽依赖关系很常见，其中较为经典的操作为 GroupByKey（将输入的 key－value

类型的数据进行分组，对于相同 Key 的 value 值进行合并，生成一个 Tuple2 数组，如图 2-3 所示)，具体代码和操作结果如下所示，这里输入 5 个 Tuple2 类型的数据，通过运行产生 3 个 Tuple2 数据。

```
def main(args:Array[String]){
  //设置输入的 Tuple2 数组
val data = Array(Tuple2("spark",100),Tuple2("spark",95),
Tuple2("hadoop",99),Tuple2("hadoop",80),Tuple2("scala",75))
  //将数组内容转化为 RDD
val rdd = sc. parallelize(data)
  //对 RDD 进行 groupByKey 操作
val rddGrouped = rdd. groupByKey()
  //输出结果
rddGrouped. collect. foreach(println)
}
```

操作结果如图 2-3 所示。

```
(scala, CompactBuffer (75))
(spark, CompactBuffer (100, 95))
(hadoop, CompactBuffer (99, 80))
```

图 2-3　GroupByKey 结果

本节内容主要讲解关于 RDD 依赖关系的基本定义，并从源代码层面讲解依赖关系的真正实现方式，但是由于整个 Spark 计算都是基于 RDD 的计算，对于 RDD 之间的依赖关系的深刻理解是很重要的一部分，希望读者可以仔细阅读 Dependency. scala 类下的源代码，并且多做几个具体的案例（推荐在 Spark - Shell 中进行），以加深对依赖关系的理解。

Section 2.2 有向无环图（Directed Acyclic Graph，DAG）

2.2.1 什么是 DAG

在图论中，如果一个有向图无法从任意顶点出发经过若干条边回到该点，则这个图就是一个有向无环图（DAG 图）。而在 Spark 中由于计算过程很多时会有先后顺序，受制于某些任务必须比另一些任务较早执行的限制，必须对任务进行排队，形成一个队列的任务集合，这个队列的任务集合就是 DAG 图。每一个顶点就是一个任务，每一条边代表一种限制约束（Spark 中的依赖关系）。

通过 DAG，Spark 可以对计算的流程进行优化，对于数据处理可以对单一结点上进行的计算操作合并，并且计算中间数据通过内存进行高效读写，对于数据处理需要涉及 Shuffle 操作的步骤进行 Stage 划分，从而使计算资源的利用更加高效、合理，减少计算资源的等待

过程，减少计算中间数据读写产生的时间浪费（基于内存的高效读写）。

2.2.2 DAG 的生成机制

Spark 中的 DAG 生成过程的重点便是对于 Stage 的划分，其划分依据是 RDD 的依赖关系，对于不同的依赖关系高层调度器会进行不同的处理。对于窄依赖，RDD 之间的数据不需要进行 Shuffle，多个数据处理可以在同一台机器的内存中完成，所以窄依赖在 Spark 中被划分为同一个 Stage；对于宽依赖，由于 Shuffle 的存在，必须等到父 RDD 的 Shuffle 处理完成后才能开始接下来的计算，所以会在此处进行 Stage 的切分。

在 Spark 中 DAG 生成的流程关键在于回溯，在程序提交后，高层调度器将所有的 RDD 看成是一个 Stage，然后对此 Stage 进行从后往前的回溯，遇到 Shuffle 就断开，遇到窄依赖则归并到同一个 Stage。等到所有的步骤回溯完成后，便生成了一个 DAG 图。

DAG 生成的相关源代码位于 Spark 的 DAGSchueduler. scala 的 getParentStages 方法中，方法具体内容如下所示，源代码中通过 for 循环遍历 RDD 依赖关系，并对其进行模式匹配，如果关系为 Shuffle Dependency，则将其加入到 ShuffleMapStage 中。

```
private def getParentStages(rdd:RDD[_],firstJobId:Int):List[Stage] = {
  val parents = new HashSet[Stage]
  val visited = new HashSet[RDD[_]]
  val waitingForVisit = new Stack[RDD[_]]
  //定义 visit 函数
  def visit(r:RDD[_]) {
  //判断该 RDD 是否被访问
    if (!visited(r)) {
  //如果未被访问,则再加入到 visited 数据结构中
      visited + = r
    //将该 RDD 的依赖关系进行循环遍历
      for (dep < - r.dependencies) {
        //对依赖关系进行模式匹配
        dep match {
        //如果是宽依赖,则形成一个新的 shuffleMapStage
          case shufDep:ShuffleDependency[_,_,_] =>
            parents + = getShuffleMapStage(shufDep,firstJobId)
        //如果不是,则将此 RDD 放入堆栈
          case _ =>
            waitingForVisit.push(dep.rdd)
        }
      }
    }
  }
  //如果此 RDD 已被访问,则直接放入堆栈
```

```
    waitingForVisit.push(rdd)
    while (waitingForVisit.nonEmpty){
      visit(waitingForVisit.pop())
    }
    //返回 stage 的列表
    parents.toList
}
```



2.2.3 DAG 的逻辑视图

本节主要通过一个简单计数案例，从中讲解 DAG 具体生成的流程和关系。示例代码如下。

```
val conf = new SparkConf()//create SparkConf
conf.setAppName("Wow,My First Spark App")//set app name
conf.setMaster("local")//run local
val sc = new SparkContext(conf)
val lines = sc.textFile("C://Users//feng//IdeaProjects//WordCount//src//SparkText.txt",1)
//操作 1 flatMap 由 lines 通过 flatMap 形成新的 MapPartitionRDD
val words = lines.flatMap{lines => lines.split(" ")}
//操作 2 map 由 word 通过 Map 操作形成新的 MapPartitionRDD
val pairs = words.map {word => (word,1)}
//操作 3 reduceByKey(包含 2 步 reduce)
//此步骤生成 MapPartitionRDD 和 ShuffleRDD
val WordCounts = pairs.reduceByKey(_ + _)
WordCounts.collect.foreach(println)
sc.stop()
```

在程序正式运行前，Spark 的 DAG 调度器会将整个流程设定为一个 Stage，此 Stage 包含 3 个操作和 4 个 RDD，分别为 MapPartitionRDD（读取文件数据时）、MapPartitionRDD（flatMap 操作）、MapPartitionRDD（map 操作）、MapPartitionRDD（reduceByKey 的 local 段的操作）和 ShuffleRDD（reduceByKeyshuffle 操作）。

1）回溯整个流程，在 shuffleRDD 与 MapPartitionRDD（reduceByKey 的 local 段的操作）中存在 Shuffle 操作，整个 RDD 先在此切开，形成两个 Stage。

2）继续向前回溯，MapPartitionRDD（reduceByKey 的 local 段的操作）与 MapPartitionRDD（map 操作）中间不存在 Shuffle 操作（即两个 RDD 的依赖关系为窄依赖），归为同一个 Stage。

3）继续回溯，发现之前的所有 RDD 之间都不存在 Shuffle，应归为同一个 stage。

4）回溯完成，形成 DAG，由两个 Stage 构成。

- 第一个 Stage 由 MapPartitionRDD（读取文件数据时）、MapPartitionRDD（flatMap 操作）、MapPartitionRDD（map 操作）和 MapPartitionRDD（reduceByKey 的 local 段的操

作构成，如图 2-4 所示。

- 第二个 Stage 由 ShuffleRDD（reduceByKey shuffle 操作）构成，如图 2-5 所示。

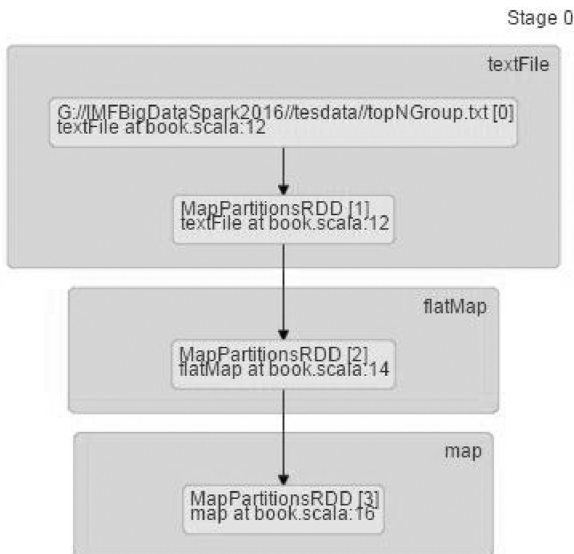


图 2-4　Stage0 的构成

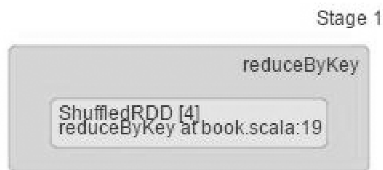


图 2-5　Stage1 的构成

Section 2.3 RDD 内部的计算机制

本节开始讲解 RDD 运算基本单位 Task，目的是让读者可以从根本上了解 sparkRDD 计算究竟是如何进行的，计算流程是如何处理的等内容。

2.3.1 RDD 的计算任务（Task）

1. 什么是 Task

Task 是计算运行在集群上的基本计算的单位。一个 Task 负责处理 RDD 的一个 Partition，一个 RDD 的多个 Patition 会分别由不同的 Task 去处理，通过之前对于 RDD 的窄依赖关系的讲解可以发现，在 RDD 的窄依赖中，子 RDD 中 Partition 的个数基本都大于等于父 RDD 中 Partition 的个数，所以 Spark 计算中对于每一个 Stage 分配的 Task 的数目是基于该 Stage 中最后一个 RDD 的 Partition 个数来决定的。最后一个 RDD 如果有 100 个 Partition，则 Spark 对这个 Stage 分配 100 个 Task。

Task 运行于 Executor 之上，而 Execuotr 位于 CoarseGrainedExecutorBackend（JVM 进程）中。

2. Task 的分类

Spark Job 中，根据 Task 所处 Stage 的位置可将 Task 分为两类，第一类为 shuffleMapTask，指 Task 所处的 Stage 不是最后一个 Stage，也就是 Stage 的计算结果还没有输出，而是通过 Shuffle 交给下一个 Stage 使用；第二类为 resultTask，指 Task 所处的 Stage 是 DAG

中最后一个 Stage，也就是 Stage 计算结果需要进行输出等操作，计算到此为止已经结束。简单地说，Spark Job 中除了最后一个 Stage 的 Task 称为 resultTask 外，其他所有的 Task 都称为 shuffleMapTask。

2.3.2 RDD 的计算过程

Spark 中的 Job 是由具体的 Task 构成的，基于 Spark 程序内部的调度模式，根据宽依赖的关系划分不同的 Stage，最后一个 Stage 依赖倒数第二个 Stage 等，从最后一个 Stage 获取结果；在 Stage 内部有一系列的任务，这些任务被提交到集群上的计算结点进行计算，计算结点执行计算逻辑时，复用位于 Executor 中的线程池中的线程，线程中运行的任务是调用具体 Task 的 run 方法进行计算，此时，如果调用具体 Task 的 run 方法，需要考虑不同 Stage 内部具体 Task 的类型，Spark 规定最后一个 Stage 中的 Task 的类型称为 resultTask，因为需要获取最后的结果，前面所有 Stage 的 Task 都是 shuffleMapTask。

RDD 在进行计算前，Driver 给 Executor 发送消息，让 Executor 启动 Task，在 Executor 成功启动 Task 后，通过消息机制汇报启动成功信息给 Driver。如下图 2-6 所示。

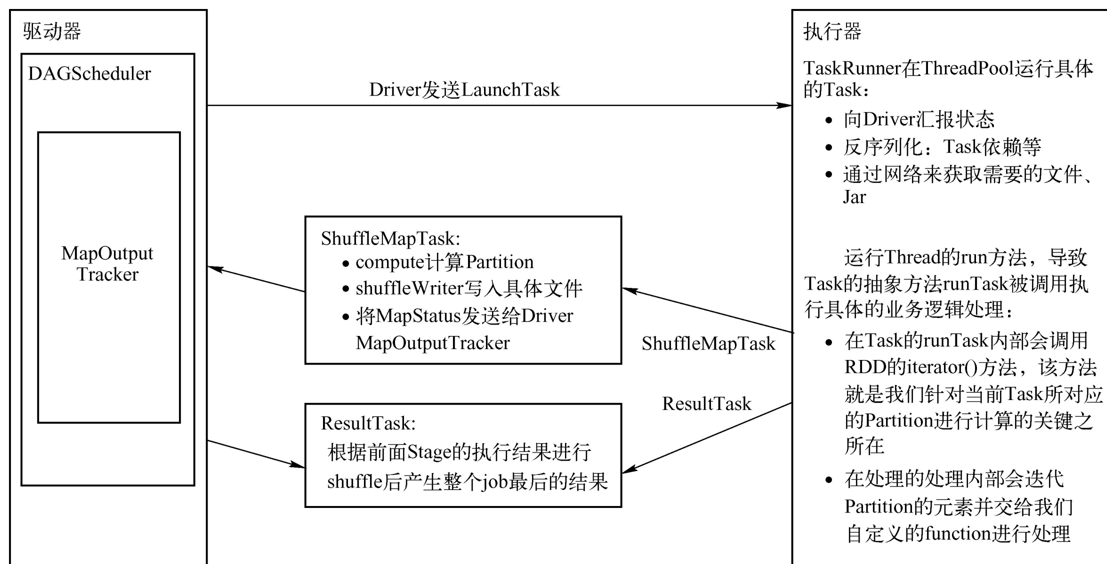


图 2-6　RDD 的计算过程

具体操作如下。

1）Driver 中的 CoarseGrainedSchedulerBackend 给 CoarseGrainedExecutorBackend 发送 LaunchTask 消息。

2）首先反序列化 TaskDescription（下述第 11 行）。

```
1.  override def receive:PartialFunction[Any,Unit] = {
2.      case LaunchTask(data) =>
3.      /**
4.  如果无可用于计算的 Executor,则输出错误日志
5.       **/
```

```
6.            if (executor == null) {
7.              logError("Received LaunchTask command but executor was null")
8.              System.exit(1)
9.            } else {
10.             //反序列化 TaskDescription
11.             val taskDesc = ser.deserialize[TaskDescription](data.value)
12.             //写入日志
13.   logInfo("Got assigned task " + taskDesc.taskId)
14. //调用 executor 的 launchTask,分配线程给 Task
15.             executor.launchTask(this,taskId = taskDesc.taskId,attemptNumber
16.                 = taskDesc.attemptNumber,taskDesc.name,taskDesc.serializedTask)
17. }
18. }
```

3）Executor 会通过 LaunchTask 来执行 Task（上述第 15 行）。

4）Executor 的 launchTask 方法中通过 taskRunner 对象在 threadPool 运行具体的 Task（下述代码第 9 行）。

```
1.  //代码来源 Executor.scala
2.  def launchTask(
3.            context:ExecutorBackend,
4.            taskId: Long,
5.            attemptNumber: Int,
6.            taskName: String,
7.            serializedTask: ByteBuffer): Unit = {
8.        //调用 TaskRunner 句柄创建 TaskRunner 对象
9.          val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attemptNumber,
           taskName,
10.           serializedTask)
11.         //将创建的 taskRunner 对象放入即将进行的堆栈中
12.         runningTasks.put(taskId,tr)
13.         //从线程池中分配一条线程给 taskRunner
14.         threadPool.execute(tr)
15.    }
```

在 taskRunner 的 run 方法中首先会通过 statusUpdate 给 Driver 发信息汇报自己的状态，说明自己是 running 状态（下述代码第 9 行，第 22 行）。

同时 TaskRunner 内部会做一些准备工作，例如反序列化 Task 的依赖（下述代码第 13 行），通过网络获取需要的文件 Jar 等（下述代码第 13 行）。

之后反序列化 Task 本身（下述代码第 15 行）。

```
1.  //代码来源 Executor.scala
2.  override def run():Unit = {
```

```
3.     val taskMemoryManager  =  new TaskMemoryManager( env. memoryManager, taskId)
4.   val deserializeStartTime  =  System. currentTimeMillis( )
5.   Thread. currentThread. setContextClassLoader( replClassLoader)
6.   val ser  =  env. closureSerializer. newInstance( )
7.   logInfo( s" Running  $ taskName( TID  $ taskId)" )
8.     //通过 executorBackend 的 statusUpdate 方法,给 Driver 发送 Task 的 Running 状态,第 18 行
       引用了 statusUpdate 方法
9.   execBackend. statusUpdate( taskId, TaskState. RUNNING, EMPTY_BYTE_BUFFER)
10.  var taskStart: Long =  0
11.  startGCTime  =  computeTotalGcTime( )
12.  try {
13.  val ( taskFiles, taskJars, taskBytes)  =  Task. deserializeWithDepaaendencies( serializedTask)
14.  updateDependencies( taskFiles, taskJars)
15.  task =  ser. deserialize[ Task[ Any] ] ( taskBytes, Thread. currentThread. getContextClassLoader)
16.  task. setTaskMemoryManager( taskMemoryManager)
17.
18.  //下述代码来源:CoarseGradeExecutorBackend. scala
19.  override def statusUpdate( taskId: Long, state: TaskState, data: ByteBuffer) {
20.      val msg  =  StatusUpdate( executorId, taskId, state, data)
21.      driver match {
22.        case Some( driverRef)  =>  driverRef. send( msg)
23.        case None => logWarning( s" Drop  $ msg because has not yet connected to driver" )
24.      }
25.    }
26.  }
27.  }
```



5）调用反序列化后的 Task. run 方法来执行任务，并获得执行结果（下述代码第 5 行）。

```
1.   //Task 计算开始的时间
2.   taskStart = System. currentTimeMillis( )
3.         var threwException  =  true
4.         val ( value, accumUpdates)  =  try {
5.           //运行 Task 的 run 方法
6.           val res =  task. run(
7.             taskAttemptId  =  taskId,
8.             attemptNumber  =  attemptNumber,
9.             metricsSystem  =  env. metricsSystem)
10.          threwException  =  false
11.          res
12.        } finally {
13.        //计算完成后清理内存,并检查是否有内存溢出
```

```
            val freedMemory = taskMemoryManager. cleanUpAllAllocatedMemory( )
            if( freedMemory >  0) {
              val erraMsg = s" Managed memory leak daetected; size =  $ freedMemory bytes,TID =
               $ taskId"
              if ( conf. getBoolean( " spark. unsafe. exceptionOnMemoryLeak" ,false) &&  !threwEx-
              ception) {
                throw newSparkException( errMsg)
              } else {
                logError( errMsg)
              }
            }
          }
          //计算完成的时间
          val taskFinish  =  System. currentTimeMillis( )
```

通过查看 task. run 方法的源代码可以发现，run 方法调用了 runTask 的方法，而 runTask 方法是一个抽象方法，runTask 方法内部会调用 RDD 的 iterator( )方法，该方法就是针对当前 Task 所对应的 Partition 进行计算的关键所在，在 Executor 的内部会迭代 Partition 的元素并交给用户自定义的 Function 进行处理。

```
//代码来源:Task. scala
try {
        (runTask( context) ,context. collectAccumulators( ) )
   }
```

Task 有两个子类，分别是 ShuffleMapTask 和 ResultTask，接下来分别对两者进行讲解：

① ShuffleMapTask。

```
override def runTask( context:TaskContext) :MapStatus = {
    //Deserialize the RDD using the broadcast variable.
    val deserializeStartTime = System. currentTimeMillis( )
    //创建序列化器
    val ser = SparkEnv. get. closureSerializer. newInstance( )
    //反序列化出 RDD 和依赖关系
    val (rdd,dep) = ser. deserialize[ (RDD[_],ShuffleDependency[_,_,_]) ](
      ByteBuffer. wrap( taskBinary. value) ,Thread. currentThread. getContextClassLoader)
    //RDD 反序列化的时间
    _executorDeserializeTime  =  System. currentTimeMillis( ) - deserializeStartTime
    metrics  =  Some( context. taskMetrics)
    //创建 shuffle 的 writer 对象,用来将计算结果写入 shuffle 管理器
    var writer:ShuffleWriter[ Any,Any]  =  null
    try {
```



```
15.         //实例化 shuffleManager
16.         val manager = SparkEnv.get.shuffleManager
17.         //对 writer 对象赋值
18.         writer = manager.getWriter[Any,Any](dep.shuffleHandle,partitionId,context)
19.         //将计算结果通过 writer 对象的 write 方法写入 shuffle 的过程
20.         writer.write(rdd.iterator(partition,context).asInstanceOf[Iterator[_ <: Product2[Any,
            Any]]])
21.         writer.stop(success = true).get
22.       } catch {
23.         case e:Exception =>
24.           try {
25.             if (writer != null) {
26.               writer.stop(success = false)
27.             }
28.           } catch {
29.             case e:Exception =>
30.               log.debug("Could not stop writer",e)
31.   }
32.           throw e
33.       }
34.     }
```

首先，ShuffleMapTask 会反序列化 RDD 及其依赖关系（上述代码第 7 行）。

（上述代码第 20 行）通过调用 RDD 的 iterator 方法进行计算，而 iterator 方法中进行的最终运算的方法是 compute()。

```
//代码来源:RDD.scala
final def iterator(split:Partition,context:TaskContext):Iterator[T] = {
      //判断此 RDD 的持久化等级是否为 NONE(不进行持久化)
      if (storageLevel != StorageLevel.NONE) {
        //如果持久化等级不为空,则先去缓存管理器中查看是否有缓存,若无再进行计算
        SparkEnv.get.cacheManager.getOrCompute(this,split,context,storageLevel)
      } else {
        //如果持久化等级为 NONE,则直接进行计算
        computeOrReadCheckpoint(split,context)
      }
    }
//代码来源:RDD.scala
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator
[T] =
    {
//判断该 RDD 是否进行过 checkpoint
```




```
        if (isCheckpointedAndMaterialized) {
//从上次检查点开始进行计算
            firstParent[T].iterator(split,context)
        } else {
            //直接进行计算
            compute(split,context)
        }
    }
//RDD 的子类 MapPartitionsRDD 代码
private[spark] class MapPartitionsRDD[U:ClassTag,T:ClassTag](
        prev:RDD[T],
        f:(TaskContext,Int,Iterator[T]) => Iterator[U],   //(TaskContext,partition index,iterator)
        preservesPartitioning:Boolean = false)
    extends RDD[U](prev){
    override val partitioner = if(preservesPartitioning)firstParent[T].partitioner else None
    override def getPartitions:Array[Partition] = firstParent[T].partitions
    override def compute(split:Partition,context:TaskContext):Iterator[U] =
        f(context,split.index,firstParent[T].iterator(split,context))
}
```

而 RDD 的 compute 方法是一个抽象方法，每个 RDD 都需要重写上次方法。

此时选择查看 MapPartitionsRDD 已经实现的 compute 方法，可以发现 compute 方法是通过 f 方法实现的，而 f 方法就是在创建 MapPartitionsRDD 时输入的操作函数。

注意：通过迭代器的不断叠加，将每个 RDD 的小函数合并成一个大的函数流。

然后在计算具体的 Partition 之后，会通过 shuffleManager 获得的 shuffleWriter 把当前 Task 计算的结果根据具体的 shuffleManager 实现写入到具体的文件，操作完成之后会把 MapStatus 发送给 Driver 端的 DAGScheduler 的 MapOutputTracker。

② ResultTask。Driver 端的 DAGSchueduler 的 MapOutputTracker 把 shuffleMapTask 执行的结果交给 ResultTask，ResultTask 根据前面 Stage 的执行结果进行 Shuffle 后产生整个 Job 最后的结果。

```
override def runTask(context:TaskContext):U = {
    //Deserialize the RDD and the func using the broadcast variables.
    val deserializeStartTime = System.currentTimeMillis()
    //创建序列化器
    val ser = SparkEnv.get.closureSerializer.newInstance()
    //反序列 RDD 和 func 处理函数
    val (rdd,func) = ser.deserialize[(RDD[T],(TaskContext,Iterator[T]) => U)](
      ByteBuffer.wrap(taskBinary.value),Thread.currentThread.getContextClassLoader)
```

```
9.          _executorDeserializeTime = System. currentTimeMillis( ) – deserializeStartTime
10.
11.         metrics = Some( context. taskMetrics)
12.         func( context, rdd. iterator( partition, context) )
13.     }
```

ResultTask 的 runTask 方法中（第 12 行）反序列化生成 func 函数，最后通过 func 函数计算出最终结果。

6）上述计算完成后，对数据进行传输。

Section 2.4 RDD 中缓存的适用场景和工作机制

2.4.1 缓存的使用

Spark 的缓存通过 RDD 的 cache 方法实现，Cache 方法是 Persist 持久化时 Storagelevel 参数默认设置为 MEMORY_ONLY。在调用 Cache 方法的时候，Cache 方法先执行不带参数的 Persist 方法，而无参数的 Persist 方法在实现时会调用默认参数为 MEMORY_ONLY 的 Persist 方法，在这个方法中，首先设置 RDD 的 Storagelevel，然后在 sparkContext 中进行标记，而且 Storagelevel 只能被设置一次，如果 Storagelevel 从 None 修改为一个值以后，就不能再修改。在 RDD 的核心计算函数迭代器函数中，当 Task 运行的时候会调用 RDD 的 Compute 方法进行计算，而 Compute 方法会调用 Iterator 方法，如果 StorageLevel 不是 NONE，那 RDD 就通过 CacheManager 的 getOrCompute 来调用获取缓存数据或者重新计算。

```
final def iterator( split: Partition, context: TaskContext) : Iterator[ T] = {
if ( storageLevel != StorageLevel. NONE) {
SparkEnv. get. cacheManager. getOrCompute( this, split, context, storageLevel)
    } else {
computeOrReadCheckpoint( split, context)
    }
}
```

对于 RDD 的缓存触发，本节通过一个基于 spark – shell 环境下的 WordCount 实例进行说明。实例代码（无缓存情况）如图 2–7 所示，运行结果如图 2–8 所示，整个任务运行时间超过 40 s，亦即如果整个任务的计算数据产生丢失，再次计算需要花费 40 s 的时间，而基于内存的数据的丢失十分常见，对此，Spark 开发了自有的持久化机制 Persist，而默认的 Persist 机制就是 Cache，也就是通过缓存实现的。

```
scala>  val wordcount = sc.textFile("/library/wordcount/input/licenses").flatMa
p(_.split(" ")).map(word=>(word,1)).reduceByKey(_+_).filter(pair=>pair._2>20).c
ollect().foreach(println)
```

图 2-7　无缓存的 WordCount 代码

```
tasks have all completed, from pool
16/02/12 18:44:20 INFO scheduler.DAGScheduler: ResultStage 1 (collect at <conso
le>:27) finished in 0.833 s
16/02/12 18:44:20 INFO scheduler.DAGScheduler: Job 0 finished: collect at <cons
ole>:27, took 40.526622 s
```

图 2-8　无缓存代码执行耗时

之后通过对代码进行改进，在计算完成后先进行缓存，然后对缓存结果进行收集和显示，可以发现，从缓存中收集数据并显示花费的时间为 1.8 s，从容错角度讲，1.8 s 的时间对比之前重新计算需要 40 s 的时间来说，已经大大降低了计算资源的浪费。

图 2-9 ~ 图 2-11 所示为对代码进行改进，使之先进行缓存后再进行计算，并显示缓存后的计算时间等信息。

```
scala>  val wordcount_caches = sc.textFile("/library/wordcount/input/licenses")
.flatMap(_.split(" ")).map(word=>(word,1)).reduceByKey(_+_).filter(pair=>pair._
2>20).cache()
```

图 2-9　对 WordCount 结果进行缓存

```
scala> wordcount_caches.collect.foreach(println)
```

图 2-10　输出缓存后的结果

```
tasks have all completed, from pool
16/02/12 18:46:53 INFO scheduler.DAGScheduler: ResultStage 5 (collect at <conso
le>:30) finished in 0.664 s
16/02/12 18:46:53 INFO scheduler.DAGScheduler: Job 2 finished: collect at <cons
ole>:30, took 1.832612 s
```

图 2-11　缓存获取计算数据耗时

2.4.2　缓存的适用场景

缓存并不是每一个步骤都需要使用的，虽然说缓存可以放置在内存中（也有可能在磁盘中），读写速度极快，但是，大量的缓存读写也会对整个集群的计算性能造成影响，如何更好地设置缓存对于一个集群的计算性能的提升也起着非常重要的作用。

以下是几种比较适合使用缓存的情况。

1）获取大量的数据之后。比如来自从外部数据源（HDFS、HIVE 和 HBASE 等），或者来自上一个 Stage，如果数据丢失需要进行重新获取或计算。

2）进行一个非常长的计算链条。如果在长计算链条中某一步骤出现数据丢失，则需要

重新计算整个链条，例如，一个1000步骤计算的计算链条，在第900步设置缓存，则如果在901步出现错误，则不需要重新计算前900个步骤，直接从缓存点（第900步）的缓存开始下一步计算。

3）某个步骤计算特别耗时。此步骤计算完成后进行缓存，以免后续的步骤出现错误需要再次进行长时间的计算。

4）进行Checkpoint之前，一般也会进行缓存。

2.4.3 缓存工作机制解析

1. 缓存的工作机制

Spark中对于缓存机制的另外一个说法为持久化，可以用Persist表示，通常Spark中用persist()或cache()方法来标记一个要被持久化的RDD，然后这个RDD一旦首次被一个动作（Action）触发计算，它将会被保留在计算结点的内存中并重用。而缓存作为一种保存数据的方式，也会出现数据丢失的可能，如果RDD的任一分区丢失了，通过使用原先创建它的转换操作，它将会被自动重算（不需要全部重算，只计算丢失的部分）。

Spark中通过unpersistRDD()来删除缓存。

每一个RDD都可以用不同的保存级别进行保存，从而允许持久化数据集在硬盘，或者在内存中作为序列化的Java对象（节省空间），甚至于跨结点复制。这些等级选择是通过将一个org. apache. spark. storage. StorageLevel对象传递给persist()方法来确定的。

2. Persist与Cache的区别

查看下述代码，可以发现cache()方法实质上是参数为memory_only的persist方法，意为仅存储在内存中，内存中无法储存的部分则不缓存，在后续再次被使用时重新计算未缓存的数据。

```
//默认的持久化等级只保存在内存中
 /** Persist this RDD with the default storage level(`MEMORY_ONLY`).  */
   def persist():this.type = persist(StorageLevel.MEMORY_ONLY)

    /** Persist this RDD with the default storage level(`MEMORY_ONLY`).  */
    def cache():this.type = persist()
```

3. Unpersist操作

对于清除缓存的操作，源代码如下所示，就是将Persist的持久化级别改为NONE。

```
/**
    * Mark the RDD as non-persistent,and remove all blocks for it from memory and disk.
    *
    * @param blocking Whether to block until all blocks are deleted.
    * @return This RDD.
    */
def unpersist(blocking:Boolean = true):this.type = {
```

```
//输出日志
logInfo("Removing RDD " + id + " from persistence list")
//移除已持久化的数据
sc.unpersistRDD(id,blocking)
//设置持久化等级为 NONE
storageLevel = StorageLevel.NONE
this
}
```

4. 持久化的保存等级 StorageLevel

Spark 中对持久化等级定义分为 12 类，源代码内容如下。

```
object StorageLevel {
  val NONE = newStorageLevel(false,false,false,false)
  val DISK_ONLY = newStorageLevel(true,false,false,false)
  val DISK_ONLY_2 = newStorageLevel(true,false,false,false,2)
  val MEMORY_ONLY = newStorageLevel(false,true,false,true)
  val MEMORY_ONLY_2 = newStorageLevel(false,true,false,true,2)
  val MEMORY_ONLY_SER = newStorageLevel(false,true,false,false)
  val MEMORY_ONLY_SER_2 = newStorageLevel(false,true,false,false,2)
  val MEMORY_AND_DISK = newStorageLevel(true,true,false,true)
  val MEMORY_AND_DISK_2 = newStorageLevel(true,true,false,true,2)
  val MEMORY_AND_DISK_SER = newStorageLevel(true,true,false,false)
  val MEMORY_AND_DISK_SER_2 = newStorageLevel(true,true,false,false,2)
  val OFF_HEAP = newStorageLevel(false,false,true,false)
}
```

- MEMORY_ONLY 表示数据只存储在内存中，如果不能被内存装下，不能装下的那部分分区不被缓存，并在需要时重新计算。此模式为 Spark 持久化的默认模式。
- MEMORY_AND_DISK 表示如果 RDD 数据不能被内存装下，则超出的分区会被保存在硬盘上，并在需要时读取。
- MEMORY_ONLY_SER 表示对象正在存储中（每一分区占用一个字节数组），通常来说，这样对于控件的利用率更加高，尤其是在使用 fast serizlizer 时，但是读取时会比较占用 CPU。
- MEMORY__AND_DISK_SER 与 MEMORY_ONLY_SER 类似，但是会把超出内存的分区记录在磁盘。
- DISK_ONLY 表示 RDD 数据只储存在磁盘。
- MEMORY_ONLY_2 每一个分区都储存在两个集群结点的内存中上。
- MEMPRY_AND_DISK_2 每一个分区都储存在两个集群结点的内存和磁盘中。

Section 2.5 RDD 的检查点（Checkpoint）的适用场景和工作机制

Spark 中对于数据保存除了持久化操作之外还存在一种检查点（Checkpoint）方式，原因在于缓存的方式虽然也可以以文件形式保存在磁盘中，但是磁盘会出现损坏，文件也会出现丢失。对于此类危险，Hadoop 的 HDFS 多备份机制起到了很好的容错能力，Spark 的外部数据源可以是 HDFS，而 HDFS 对于文件的保存容错能力又很强大，便出现了 Checkpoint 机制，Checkpoint 可以将保存的数据以文件的方式保存在 HDFS 上，借用 HDFS 强大的文件容错能力，从而降低数据被破坏或者丢失的风险。

2.5.1 Checkpoint 的触发

对于 Checkpoint，本节依旧通过 spark－shell 中的 WordCount 实例进行解释。

对要进行 Checkpoint 的数据先进行缓存（见图 2-12），之后再进行 Checkpoint 操作（见图 2-13），最后对 Checkpoint 后的对象进行收集和输出（见图 2-14）。同时，通过查看 HDFS 上的文件，可以找到进行 Checkpoint 后保存的文件（见图 2-15），至此完成一个最简单的 Checkpoint 实例。

```
scala> val wordcount= sc.textFile("/library/wordcount/input/licenses").flatMap(
_.split(" ")).map(word=>(word,1)).reduceByKey(_+_).cache
```

图 2-12　缓存数据

```
scala> wordcount.checkpoint
```

图 2-13　进行 checkpoint 操作

```
scala> wordcount.filter(pair=>pair._2>20).collect().foreach(println)
```

图 2-14　筛选 checkpoint 数据后输出

Browse Directory

/library/sparkCheckPoint/29e37500-f88d-4e43-8a8e-1575fded3817　Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	root	supergroup	0 B	0	0 B	rdd-19
drwxr-xr-x	root	supergroup	0 B	0	0 B	rdd-31
drwxr-xr-x	root	supergroup	0 B	0	0 B	rdd-38

图 2-15　Checkpoint 保存在 HDFS 的文件

2.5.2 Checkpoint 的适用场景

缓存的使用虽然广泛，但是对于很多的数据分析算法，使用者需要更加安全的数据缓存机制，以最大程度地确保数据的安全。

- 机器学习或者图计算等对应于数据复用十分频繁和重复迭代次数较多的应用程序。
- 对于数据保存的安全化级别也特别高，例如流计算中需要实时统计过去某段时间的数据，需要数据有极高的安全化等级。

以上情况都可以通过 checkpoint 借用 HDFS 的高容错特性来增强数据的安全性，同时也减少了数据重新计算时的开销。

2.5.3 Checkpoint 工作机制解析

1. Checkpoint 的创建

Spark 关于 Checkpoint 的源代码位于 RDD. Scala 文件中的 Checkpoint 方法，具体内容如下所示。从中可以看出在创建 Checkpoint 时，Spark 会先判断是否有 CheckpointDir 的地址（存放 Checkpoint 数据内容文件的地址），若不存在则报错；若存在再检测已有的 Checkpoint 信息，看是否存在 Checkpoint，如果没有，则创建 CheckpointData，其中包含了 Checkpoint 的信息。

```
def checkpoint( ):Unit = RDDCheckpointData. synchronized {
    // NOTE:we use a global lock here due to complexities downstream with ensuring
    // children RDD partitions point to the correct parent partitions. In the future
    // we should revisit this consideration.
    //判断是否定义了 checkpoint 文件的保存地址
    if ( context. checkpointDir. isEmpty) {
    //如过没有定义,则返回异常,未定义 checkpoint 文件夹
      throw newSparkException( "Checkpoint directory has not been set in the SparkContext" )
    } else if ( checkpointData. isEmpty) {
      //创建 checkpointData,其中包含了 checkpoint 的信息
      checkpointData = Some( new ReliableRDDCheckpointData( this) )
  }
}
```

在 Spark 中，某 RDD 进行 Checkpoint 操作后会将此 RDD 的依赖关系清空，此项变化在算法计算中具有极大的意义，假设一个运算步骤有 100 步（同一个 Stage 中），Spark 框架会在计算时将这 100 步的函数合并为一个大函数，然后进行计算，而如果在第 90 步处进行了 Checkpoint，则 Spark 计算时将前 90 步函数合并为一个大的函数串进行计算，并在计算完成时进行保存，之后 10 步的函数合并为一个大的函数串进行计算。如果出现错误，只需要对后 10 步进行计算，具体源代码如下所示：第 10 行中便是 RDD 进行 Checkpoint 时会调用 rdd. markCheckpointed( ) 方法，而此方法将 RDD 的 dependency 进行了清空操作。



```
valnewRDD = doCheckpoint( )

  // Update our state and truncate the RDD lineage
  RDDCheckpointData. synchronized {
    //设置保存的数据
    cpRDD = Some( newRDD)
    //设置状态
    cpState = Checkpointed
//清理依赖关系
    rdd. markCheckpointed( )
   }

 private[ spark] def markCheckpointed( ) :Unit = {
   clearDependencies( )
   partitions_ = null
   deps = null     // Forget the constructor argument for dependencies too
 }

protected def clearDependencies( ) {
   //依赖关系清空
   dependencies_ = null
 }
```

2. Checkpoint 的写入

写入数据至 Checkpoint 文件的源代码位于 ReliableCheckpointRDD. Scala 的 writeRDDToCheckpointDirectory( )。如下面的源代码所示，分别完成对于 checkpointDir 目录下的文件的创建和具体内容的写入。

```
def writeRDDToCheckpointDirectory[T:ClassTag](
      originalRDD:RDD[T],
      checkpointDir:String,
      blockSize:Int = -1):ReliableCheckpointRDD[T] = {

    val sc = originalRDD. sparkContext

    //根据 checkpoint 的设置路径创建文件夹
    val checkpointDirPath = new Path( checkpointDir)
    val fs = checkpointDirPath. getFileSystem( sc. hadoopConfiguration)
    //判断文件夹是否创建成功
    if ( !fs. mkdirs( checkpointDirPath) ) {
      throw newSparkException( s"Failed to create checkpoint path$checkpointDirPath" )
    }
```

```

    //将 Hadoop 配置文件反序列化后以广播的方式创建
        val broadcastedConf = sc. broadcast(
          new SerializableConfiguration( sc. hadoopConfiguration) )
    //运行一个新 job,将需要进行 checkpoint 的文件写入创建的 checkpoint 文件夹中
        sc. runJob( originalRDD,
          writePartitionToCheckpointFile[ T] ( checkpointDirPath. toString, broadcastedConf) _)

        if ( originalRDD. partitioner. nonEmpty) {
          writePartitionerToCheckpointDir( sc, originalRDD. partitioner. get, checkpointDirPath)
        }

        val newRDD = new ReliableCheckpointRDD[ T] (
          sc, checkpointDirPath. toString, originalRDD. partitioner)
        //判断之前的两次内容是否一致,确保写入的文件正确无误
        if ( newRDD. partitions. length != originalRDD. partitions. length) {
          throw newSparkException(
            s" Checkpoint RDD $newRDD(${ newRDD. partitions. length} ) has different " +
              s" number of partitions from original RDD $originalRDD
              (${ originalRDD. partitions. length} ) " )
        }
        newRDD
    }
```

3. Checkpoint 的触发

当需要对已进行了 Checkpoint 的数据进行处理时，框架默认运行 docheckpoint 方法（如下所示）。若当前 RDD 的 checkpointData 是被定义的（此 RDD 已被进行 Checkpoint），则获得数据；若当前 RDD 的 checkpointData 是未被定义的，则按照依赖遍历 RDD 查询此 RDD 依赖 RDD 是否存在 Checkpoint。

```
private[ spark] def doCheckpoint( ): Unit = {
    RDDOperationScope. withScope( sc, " checkpoint" , allowNesting = false, ignoreParent = true) {
      if ( !doCheckpointCalled) {
        doCheckpointCalled = true
        //判断当前 RDD 是否是被标记的 checkpointRDD
        if ( checkpointData. isDefined) {
          //直接获取 checkpoint 数据
          checkpointData. get. checkpoint( )
        } else {
          //根据依赖关系向前回溯查找到进行 checkpoint 的 RDD,并获取 checkpoint 数据
            dependencies. foreach( _. rdd. doCheckpoint( ) )
        }
```

```
        }
    }
}
```

Section 2.6 RDD 容错原理及其四大核心要点

2.6.1 RDD 容错原理

由于 RDD 不同的依赖关系，导致 Spark 对于不同的依赖关系有不同的处理方式。

- 对于宽依赖而言，由于宽依赖实质是指父 RDD 的一个分区会对应一个子 RDD 的多个分区，在此情况下出现部分计算结果的丢失，单一计算丢失的数据无法达到效果，便重新计算该步骤所有的数据，从而会导致计算数据的重复。
- 对于窄依赖而言，由于窄依赖实质是指父 RDD 的分区最多被一个子 RDD 所使用，在此情况下出现部分计算的错误，与之计算相关只是出错部分所依赖的父 RDD 相关部分的数据，不需要重新计算所有数据，只需要重新计算出错部分的数据。

2.6.2 RDD 容错的四大核心要点

对于 Spark 框架层面的容错机制，主要可以分为三大层面，分别为调度层（包含 DAG 生成和 Task 重算两大核心）、RDD 血统层和 Checkpoint 层。其中包含 SparkRDD 容错的 4 大核心要点：

- Stage 输出失败，上层调度器 DAGScheduler 重试；
- Spark 计算中 Task 内部任务失败，底层调度器重试；
- RDD Lineage 血统中窄依赖、宽依赖计算；
- Checkpoint 缓存。

1. 调度层

从调度层面讲，错误主要出现在两个方面，分别是在 Stage 输出时出错和计算时出错。

（1）DAG 生成层

Stage 输出失败，上层调度器 DAGScheduler 会进行重试，如下面的源代码所示。

```
/**
  * Resubmit any failed stages. Ordinarily called after a small amount of time has passed since
  * the last fetch failure.
  */
 private[scheduler] def resubmitFailedStages() {
    //判断是否存在失败的 Stages
    if (failedStages.size > 0) {
```

```
            // Failed stages may be removed by job cancellation,so failed might be empty even if
            // the ResubmitFailedStages event has been scheduled.
            //输出日志
            logInfo("Resubmitting failed stages")
            clearCacheLocs()
            //获取所有失败 Stage 的列表
            val failedStagesCopy = failedStages.toArray
            //清空 failedStages
            failedStages.clear()
            //对之前获取的所有失败的 Stage 根据 jobId 进行排序后进行逐一的重试
            for(stage <- failedStagesCopy.sortBy(_.firstJobId)) {
              submitStage(stage)
            }
          }
        submitWaitingStages()
    }
```

（2）Task 计算层

Spark 计算过程中，计算内部某个 Task 任务出现失败，底层调度器会对此 Task 进行若干次的重试（默认为 4 次，如下述代码第 2 行所示）。以下为相关信息的源代码。

```
1.  private[spark] classTaskSchedulerImpl() {
2.    def this(sc:SparkContext) = this(sc,sc.conf.getInt("spark.task.maxFailures",4))
3.
4.  //下列代码来源 TaskSetManager.scala
5.  def handleFailedTask(tid:Long,state:TaskState,reason:TaskEndReason) {
6.        //对于失败的 Task 的 numFailures 进行计数加 1
7.          numFailures(index) += 1
8.          //判断失败的 Task 次数是否大于设定的最大失败次数,如果大于则输出日志,并不
    再重试
9.          if (numFailures(index) >= maxTaskFailures) {
10.           logError("Task %d in stage %s failed %d times;aborting job".format(
11.           index,taskSet.id,maxTaskFailures))
12.           abort("Task %d in stage %s failed %d times,most recent failure:%s\nDriver stack-
            trace:"
13.           .format(index,taskSet.id,maxTaskFailures,failureReason),failureException)
14.         return
15.        }
16.     }
17.     //如果运行的 Task 为 0,则完成 Task 步骤
18.     maybeFinishTaskSet()
19.   }
```

2. RDD Lineage 血统层容错

Spark 中 RDD 采用高度受限的分布式共享内存，且新的 RDD 的产生只能够通过其他 RDD 上的批量操作来创建，依赖于以 RDD 的 Lineage 为核心的容错处理，在迭代计算方面比 Hadoop 快 20 多倍，同时还可以在 5 ~ 7s 内交互式的查询 TB 级别的数据集。

Spark RDD 实现基于 Lineage 的容错机制，基于 RDD 的各项 Transformation 构成了 compute chain，在部分计算结果丢失的时候可以根据 Lineage 重新计算恢复。

- 在窄依赖中，在子 RDD 的分区丢失要重算父 RDD 分区时，父 RDD 相应分区的所有数据都是子 RDD 分区的数据，并不存在冗余计算；
- 在宽依赖情况下，丢失一个子 RDD 分区重算的每个父 RDD 的每个分区的所有数据并不是都给丢失的子 RDD 分区用的，会有一部分数据相当于对应的是未丢失的子 RDD 分区中需要的数据，这样就会产生冗余计算开销和巨大的性能浪费。

3. Checkpoint 层容错

Spark Checkpoint 通过将 RDD 写入 Disk 做检查点，是 Spark lineage 容错的辅助，lineage 过长会造成容错成本过高，这时候在中间阶段做检查点容错，如果之后有结点出现问题而丢失分区，从做检查点的 RDD 开始重做 Lineage，就会减少开销。

Checkpoint 主要适用于以下两种情况：

- DAG 中的 Lineage 过长，如果重算时会开销太大，例如在 PageRank、ALS 等；
- 尤其适合于在宽依赖上做 Checkpoint，这个时候就可以避免应为 Lineage 重新计算而带来的冗余计算。

Section 2.7 通过 WordCount 实践 RDD 内部机制

2.7.1 WordCount 案例实践

本节通过 IDEA 逐步调试一个带有排序功能的 WordCount 案例，让读者了解各步骤中 RDD 的具体类型。

1. WordCount 代码

```
object WordCount {
  def main(args:Array[String]){
    val conf = new SparkConf()//create SparkConf
    conf.setAppName("Wow,My First Spark App")//设置应用程序名
    conf.setMaster("local")//设置为本地运行
    val sc = new SparkContext(conf)
    val lines = sc.textFile("C://Users//feng//IdeaProjects//WordCount//src//SparkText.txt")
    val words = lines.flatMap{lines => lines.split(" ")}
    val pairs = words.map( word => (word,1))
```

```
    val reduce = pairs.reduceByKey(_ + _)
    val sort_1 = reduce.map(pair => (pair._2, pair._1))
    val sort_2 = sort_1.sortByKey(false)
    val sort_3 = sort_2.map(pair => (pair._2, pair._1))
    val filter = sort_3.filter(pair => pair._2 > 2)
    filter.collect.foreach(wordNumberPair => println(wordNumberPair._1 + " : " + wordNumberPair._2))
    sc.stop()
  }
}
```

2. 程序使用的 SparkText.txt 文件内容

```
hadoop hadoop hadoop
spark Flink spark
scala scala object
object spark scala
spark spark
Hadoop hadoop
```

3. 程序 WordCount 调试结果

通过 IDEA 的逐步调试，会在调试窗口显示每一行代码具体操作什么类型的 RDD，以及此 RDD 通过什么依赖关系依赖于父 RDD 等重要信息（见图 2-16），程序运行结果如图 2-17 所示。

```
▶ lines = {MapPartitionsRDD@5792} "MapPartitionsRDD[1] at textFile at WordCount.scala:14"
▶ words = {MapPartitionsRDD@5806} "MapPartitionsRDD[2] at flatMap at WordCount.scala:15"
▶ pairs = {MapPartitionsRDD@5811} "MapPartitionsRDD[3] at map at WordCount.scala:16"
▶ reduce = {ShuffledRDD@5961} "ShuffledRDD[4] at reduceByKey at WordCount.scala:17"
▶ sort_1 = {MapPartitionsRDD@5968} "MapPartitionsRDD[5] at map at WordCount.scala:18"
▶ sort_2 = {ShuffledRDD@5983} "ShuffledRDD[6] at sortByKey at WordCount.scala:19"
▶ sort_3 = {MapPartitionsRDD@5992} "MapPartitionsRDD[7] at map at WordCount.scala:20"
▶ filter = {MapPartitionsRDD@6005} "MapPartitionsRDD[8] at filter at WordCount.scala:21"
```

图 2-16　调试过程图

```
spark : 5
hadoop : 5
scala : 3
```

图 2-17　WordCount 结果

2.7.2 解析 RDD 生成的内部机制

本节基于上小节程序的调试结果，逐条分析调试语句。

1. line = sc. textFile( )

本语句的作用在于从外部数据中读取数据，并生成 MapPartitionsRDD。此处需要注意：如图 2-18 所示，可以看出该 MapPartitionsRDD 的依赖为 HadoopRDD，从这里可以发现其实 textFile( )过程包含两个步骤，第一步将文件内容转化为 HadoopRDD（key – value 形式，key 为行号），第二步将 HadoopRDD 转化为 MapPartitionsRDD（value 形式，将 key – value 类型的 key 删去）。

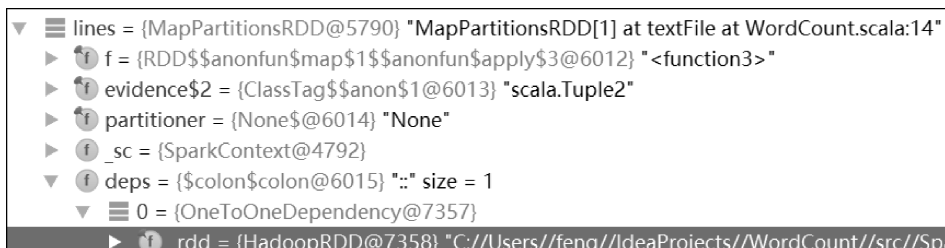


图 2-18 通过 HadoopRDD 获取数据

2. words = line. flatMap( )

此命令对于 RDD 采取 transformation（转换）操作，作用在于将 MapPartitionsRDD 中的每一个记录进行以空格为标记的切分，并把每一个 RDD 的切分结果放在一个 MapPartitionRDD 中。

3. pairs = words. map(word => (word,1))

此命令对于 RDD 采取 Transformation（转换）操作，作用在于将 MapPartitionsRDD 中的每一个记录（如 Spark（value 类型））转换为 key – value 类型（如(spark,1)），便于下一步 reduceByKey 操作。

4. reduce = pairs. reduceByKey(_ + _)

此命令对于 RDD 采取 Action（动作）操作，作用在于通过 Shuffle 将 pairs 中所有的记录按照 key 相同 value 相加的规则进行处理，并把结果放到一个 shuffleRDD 中。例如，((spark,1),(spark,1))变成((spark,2))。

同时需要注意以下两点：首先本步骤实质上分为两个步骤，第一步为 Local 级别的 Reduce，对当前计算机所拥有的数据先进行 Reduce 操作，生成 MapPartitionsRDD；第二步为 Shuffle 级别的 Reduce，基于第一步的结果，对结果进行 Shuffle – Reduce 操作，生成最终的 shuffleRDD。其次进行 Action 操作时，执行此操作之前的所有转换操作，所以调试过程中会出现此前的除 textFile 操作外的执行时间均非常短，说明 RDD 转换操作不直接进行运算。

5. sort_1 = reduce. map(pair => (pair. _2, pair. _1))

此命令对于 RDD 采取 Transformation（转换）操作，作用在于将 shuffleRDD 中的每一个记录的 key 和 value 互换，生成一个新的 MapPartitionsRDD。例如，(spark,2)变为(2,spark)

6. sort_2 = sort_1. sortByKey (false)

此命令对于 RDD 采取 Action（动作）操作，作用在于将 MapPartitionsRDD 根据 key 进行排序，并生成 shuffleRDD。

7. sort_3 = sort_2. map (pair => (pair. _2 , pair. _1))

此命令对于 RDD 采取 Transformation（转换）操作，作用在于将 shuffleRDD 中的每一个记录的 key 和 value 互换，生成一个新的 MapPartitionsRDD。例如，(2,spark)变为(spark,2)。

8. filter = sort_3. filter (pair => pair. _2 > 2)

此命令对于 RDD 采取 Transformation（转换）操作，作用在于根据 value 值筛选 MapPartitionsRDD 中的数据，输出 value 大于 2 的记录。

9. 打印输出

最后通过 collect( ) 方法将结果收集后，使用 foreach( ) 方法遍历数据并通过 println( ) 方法打印出所有数据。

Section 2.8 小结

本章主要讲解了以下内容：RDD 之间的宽窄依赖关系的形成原因，依赖关系的作用，以及其对于整体框架的容错的效果；并对基于 RDD 依赖关系产生 DAG 图的机制进行了基本讲解；之后通过对 TaskSchedulerImpl 的解析，详细解释了 RDD 如何进入运算，如何得出结果，以及在运算过程中进行持久化的原理和方式。

通过本章内容的学习，可以了解 SparkRDD 的本质和 Spark 计算框架的内容。



第 3 章　部署模式（Deploy）解析

Section 3.1 部署模式概述

在 Spark 官网部署页面（http://spark.apache.org/docs/latest/cluster-overview.html）中，可以看到当前集群支持以下 3 种集群管理器（Cluster Manager）。

1）Standalone：Spark 原生的简单集群管理器，使用 Standalone 可以很方便地搭建一个集群。

2）Apache Mesos：一个通用的集群管理器，可以在上面运行 Hadoop MapReduce 和一些服务型的应用。

3）Hadoop YARN：在 Hadoop 2 中提供的资源管理器。

另外，Spark 提供的 EC2 启动脚本，可以很方便地在 Amazon EC2 上启动一个 Standalone 集群。

实际上，除了上面这些通用的集群管理器外，Spark 内部也提供了一些方便用户测试和学习的简单集群部署模式。为了更全面地理解这些内容，本节会从 Spark 应用程序部署点切入，也就是从提交一个 Spark 应用程序开始，引出并详细解析各种部署模式。

说明：下面涉及类的描述时，如果可以通过类名唯一确定一个类的话，将直接给出类名；如果不能，会先给出全路径的类名，然后在不出现歧义的地方再简写为类名。

Section 3.2 应用程序的部署

3.2.1 应用程序部署的脚本解析

为了简化应用程序提交的复杂性，Spark 提供了各种应用程序提交的统一入口，即 spark-submit 脚本，应用程序的提交都间接或直接地调用了该脚本。下面简单分析几个脚本，包含：./bin/pyspark、./bin/sparkR、./bin/spark-shell、./bin/spark-sql、./bin/run-example，以及所有脚本最终都调用到的一个执行 Java 类的脚本./bin/spark-class。

1. 脚本 spark-shell

通过该脚本可以打开使用 Scala 语言进行开发和调试的交互式界面，脚本代码如下。

```
1. …
2. function main( ) {
3. …
4. "${SPARK_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name "Spark shell" "$@"
5. stty icanon echo >/dev/null 2>&1
6. else
7. export SPARK_SUBMIT_OPTS
8. "${SPARK_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name "Spark shell" "$@"
9. fi
10. }
11. …
```

对应第 4 行和第 8 行代码，调用了应用程序提交脚本 spark-submit。脚本 spark-shell 的基本用法如下。

```
"Usage:./bin/spark-shell [options]"
```

其他的脚本类似，下面分别针对各个脚本的用法（具体用法可查看脚本的帮助信息，比如通过 --help 选项来获取）与关键执行语句等进行简单解析。

说明：要了解如何使用工具（如脚本），最根本的是先查看其帮助信息，然后在此基础上进行扩展。

---

2. 脚本./bin/pyspark

通过该脚本可以打开使用 Python 语言开发和调试的交互式界面。

1）该脚本的用法如下。

```
"Usage:./bin/pyspark [options]"
```

2）该脚本的执行语句如下。

```
exec "${SPARK_HOME}"/bin/spark-submit pyspark-shell-main --name "PySparkShell" "$@"
```

3. 脚本./bin/sparkR

通过该脚本可以打开使用 SparkR 语言开发和调试的交互式界面。

1）该脚本的用法如下。

```
"Usage:./bin/sparkR [options]"
```

2）该脚本的执行语句如下。

```
exec "${SPARK_HOME}"/bin/spark-submit sparkr-shell-main "$@"
```



4. 脚本./bin/spark-sql

通过该脚本可以打开使用 Spark Sql 语言开发和调试的交互式界面。

1）该脚本的用法如下。

```
"Usage:./bin/spark-sql [options] [cli option]"
```

2）该脚本的执行语句如下。

```
exec "${SPARK_HOME}"/bin/spark-submit --class org.apache.spark.sql.hive.thriftserver.
SparkSQLCLIDriver "$@"
```

5. 脚本./bin/run-example

可以通过该脚本运行 Spark 自带的案例代码，该脚本中会自动补全案例类的路径。

1）该脚本的用法如下。

```
echo "Usage:./bin/run-example <example-class> [example-args]" 1>&2
echo " - set MASTER=XX to use a specific master" 1>&2
echo " - can use abbreviated example class name relative to com.apache.spark.examples" 1>&2
echo " (e.g.  SparkPi,mllib.LinearRegression,streaming.KinesisWordCountASL)" 1>&2
```

2）该脚本的执行语句如下。

```
exec "${SPARK_HOME}"/bin/spark-submit \
   --master $EXAMPLE_MASTER \
   --class $EXAMPLE_CLASS \
   "$SPARK_EXAMPLES_JAR" \
   "$@"
```

6. 脚本./bin/spark-submit

./bin/spark-submit 是提交 Spark 应用程序最常用的一个脚本。从前面各个脚本的解析可以看出，各个脚本最终都调用了./bin/spark-submit 脚本。

1）用法。该脚本的用法需要从源代码中获取，具体源代码位置参考 SparkSubmitArguments 类的方法 printUsageAndExit，代码如下。

```
val command = sys.env.get("_SPARK_CMD_USAGE").getOrElse(
  """Usage:spark-submit [options] <app jar | python file> [app arguments]
     |Usage:spark-submit --kill [submission ID] --master [spark://...]
       |Usage: spark-submit --status [submission ID] --master [spark://...]"""
     .stripMargin)
```

2）该脚本的执行语句如下。

```
exec "${SPARK_HOME}"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"
```

7. 脚本./bin/spark-class

该脚本是所有其他脚本最终都调用到的一个执行 Java 类的脚本。其中关键的执行语句如下。

```
CMD=()
while IFS=read -d'' -r ARG;do
  CMD+=("$ARG")
done << ("$RUNNER" -cp "$LAUNCH_CLASSPATH" org.apache.spark.launcher.Main "$@")
exec "${CMD[@]}"
```

其中，负责运行的 RUNNER 变量设置如下。

```
# Find the java binary
// 将 RUNNER 设置为 java
if [ -n "${JAVA_HOME}" ];then
    RUNNER="${JAVA_HOME}/bin/java"
else
    if [ `command -v java` ];then
        RUNNER="java"
    else
        echo "JAVA_HOME is not set" >&2
        exit 1
    fi
fi
```

在脚本中，LAUNCH_CLASSPATH 变量对应了 Java 命令运行时所需的 classpath 信息。而最终 Java 命令启动的类是 org.apache.spark.launcher.Main，Main 类的入口函数 main 会根据输入参数构建出最终执行的命令，即这里返回的${CMD[@]}信息，然后通过 exec 执行。

3.2.2 应用程序部署的源代码解析

本节从应用部署的角度解析相关的源代码，主要包括脚本提交时对应 JVM 进程启动的主类 org.apache.spark.launcher.Main、定义应用程序提交的行为类型的类 org.apache.spark.deploy.SparkSubmitAction、应用程序封装底层集群管理器和部署模式的类 org.apache.spark.deploy.SparkSubmit，以及代表一个应用程序的驱动程序的类 org.apache.spark.SparkContext。

1. Main 解析

从前面的脚本分析可以得出，最终都是通过 org.apache.spark.launcher.Main 类（下面简

称 Main 类）来启动应用程序的，因此，首先来分析 Main 类。

在 Main 类的源代码中，类的注释如下。



```
/**
  * Spark 启动器的命令行接口。在 Spark 脚本内部使用
  * Command line interface for the Spark launcher. Used internally by Spark scripts.
 */
```

对应在 Main 对象的入口方法 main 上的注释如下。

```
/**
  * Usage:Main [class] [class args]
  * <p>
  *命令行界面工作在两种模式下
 * This CLI works in two different modes:
  * <ul>
  *  <li>"spark-submit":if <i>class</i> is "org.apache.spark.deploy.SparkSubmit",the
  *    {@link SparkLauncher} class is used to launch a Spark application.</li>
  *   <li>"spark-class":if another class is provided,an internal Spark class is run.</li>
  * </ul>
 ...
     public static void main(String[] argsArray) throws Exception {
 ...
```

Main 类主要有两种工作模式，分别描述如下。

（1）spark-submit

启动器要启动的类为"org.apache.spark.deploy.SparkSubmit"时，对应为" spark-submit" 工作模式。此时，使用 SparkSubmitCommandBuilder 类来构建启动命令。

（2）spark-class

启动器要启动的类是除 SparkSubmit 之外的其他类时，对应为"spark-class"工作模式。此时，使用 SparkClassCommandBuilder 类的 buildCommand 方法来构建启动命令。

以"spark-submit"工作模式为例，对应的在构建启动命令的 SparkSubmitCommandBuilder 类中，上述调用的 SparkClassCommandBuilder 构造函数定义如下。

```
SparkSubmitCommandBuilder(List<String> args) {
   this.sparkArgs = new ArrayList<String>();
   List<String> submitArgs = args;
// 根据输入的第一个参数设置,包括主资源 appResource 等
   if (args.size() > 0 && args.get(0).equals(PYSPARK_SHELL)) {
     this.allowsMixedArguments = true;
     appResource = PYSPARK_SHELL_RESOURCE;
```

```
            submitArgs = args. subList(1,args. size());
        } else if (args. size() >0 && args. get(0). equals(SPARKR_SHELL)) {
            this. allowsMixedArguments = true;
            appResource = SPARKR_SHELL_RESOURCE;
            submitArgs = args. subList(1,args. size());
        } else {
            this. allowsMixedArguments = false;
        }
    …
}
```

从这里初步的参数解析可以看出，前面脚本中的参数与最终对应的主资源间的对应关系如表 3-1 所示。

表 3-1　脚本中的参数与主资源间的对应关系

脚　本　名	脚本中的参数	主　资　源
./bin/pyspark	PYSPARK_SHELL = "pyspark - shell - main"	PYSPARK_SHELL_RESOURCE = "pyspark - shell"
./bin/sparkR	SPARKR_SHELL = "sparkr - shell - main"	SPARKR_SHELL_RESOURCE = "sparkr - shell"

如果继续跟踪 appResource 赋值的源代码，可以跟踪到一些特殊类的类名与最终对应的主资源间的对应关系，部分如表 3-2 所示。

表 3-2　特殊类的类名与主资源间的对应关系

参考的脚本名	类　　名	主　资　源
./bin/spark - shell	"org. apache. spark. repl. Main"	"spark - shell"
./bin/spark - sql	"org. apache. spark. sql. hive. thriftserver. SparkSQLCLIDriver"	"spark - internal"
./sbin/start - thriftserver	"org. apache. spark. sql. hive. thriftserver. HiveThriftServer2"	"spark - internal"

有兴趣的话可以继续跟踪 SparkClassCommandBuilder 类的 buildCommand 方法的源代码，查看构建的命令具体有哪些。

通过 Main 类的简单解析，可以将前面的脚本分析结果与后面即将进行分析的 SparkSubmit 类关联起来，以便进一步解析与应用程序提交相关的其他源代码。

从前面的脚本分析可以看到，在提交应用程序时，Main 所启动的类，也就是用户最终提交执行的类是 org. apache. spark. deploy. SparkSubmit。因此下面开始解析 SparkSubmit 相关的源代码，包括提交行为的定义、提交时的参数解析，以及最终提交运行的代码解析。

2. SparkSubmitAction 解析

SparkSubmitAction 定义了提交应用程序的行为类型，源代码如下。

```
private[deploy] object SparkSubmitAction extends Enumeration {
    type SparkSubmitAction = Value
    val SUBMIT,KILL,REQUEST_STATUS = Value
}
```

从源代码中可以看到，分别定义了 SUBMIT、KILL 和 REQUEST_STATUS 这 3 种行为类型，对应提交应用、停止应用和查询应用的状态。

3. SparkSubmit 解析

SparkSubmit 的全路径为 org. apache. spark. deploy. SparkSubmit。从 SparkSubmit 类的注释可以看出，SparkSubmit 是启动一个 Spark 应用程序的主入口点，这与前面从脚本分析得到的结论是一致的。首先来看 SparkSubmit 类的注释，如下所示。

```
/**
  * 启动一个 Spark 应用程序的主入口点
 * Main gateway of launching a Spark application.
  *
  * This program handles setting up the classpath with relevant Spark
 * dependencies and provides
  * a layer over the different cluster managers and deploy modes that Spark supports.
  */
```

SparkSubmit 会帮助用户设置 Spark 相关依赖包的 classpath 设置，同时，为了帮助用户简化提交应用程序的复杂性，SparkSubmit 提供了一个抽象层，封装了底层复杂的集群管理器与部署模式的各种差异点。即，通过 SparkSubmit 的封装，集群管理器与部署模式对用户而言是透明的。

SparkSubmit 透明化（通过 SparkSubmit 的封装，集群管理器与部署模式对用户而言是透明的）的集群管理器定义的源代码如下。

```
// 集群管理器
// Cluster managers
private val YARN = 1
private val STANDALONE = 2
private val MESOS = 4
private val LOCAL = 8
private val ALL_CLUSTER_MGRS = YARN | STANDALONE | MESOS | LOCAL
```

被 SparkSubmit 透明化的部署模式定义的源代码如下。

```
// 部署模式
//Deploy modes
private val CLIENT = 1
private val CLUSTER = 2
private val ALL_DEPLOY_MODES = CLIENT | CLUSTER
```

作为提交应用程序的入口点，SparkSubmit 中根据具体的集群管理器进行参数转换、参数校验等操作，比如对模式的检查，代码中给出了针对特定情况下不支持的集群管理器与部署模式，在这些模式下提交应用程序会直接报错退出。




```
//不支持的集群管理器与部署模式
// 使用了模式匹配与守卫语句
// The following modes are not supported or applicable
(clusterManager,deployMode) match {
  case(MESOS,CLUSTER) if args.isR =>
    printErrorAndExit("Cluster deploy mode is currently not supported for R " +
      "applications on Mesos clusters.")
  case(STANDALONE,CLUSTER) if args.isPython =>
    printErrorAndExit("Cluster deploy mode is currently not supported for python " +
      "applications on standalone clusters.")
   case(STANDALONE,CLUSTER) if args.isR =>
     printErrorAndExit("Cluster deploy mode is currently not supported for R " +
      "applications on standalone clusters.")
   case(_,CLUSTER) if isShell(args.primaryResource) =>
     printErrorAndExit("Cluster deploy mode is not applicable to Spark shells.")
   case(_,CLUSTER) if isSqlShell(args.mainClass) =>
     printErrorAndExit("Cluster deploy mode is not applicable to Spark SQL shell.")
   case(_,CLUSTER) if isThriftServer(args.mainClass) =>
     printErrorAndExit("Cluster deploy mode is not applicable to Spark Thrift server.")
   case _ =>
}
```

首先，一个程序运行的入口点对应单例对象的 main 函数，因此在执行 SparkSubmit 时，对应的入口点是 objectSparkSubmit 的 main 函数，具体代码如下。

```
1.  // 入口点函数 main 的定义
2.  def main(args:Array[String]):Unit = {
3.    val appArgs = new SparkSubmitArguments(args)
4.   …
5.   // 根据 3 种行为分别进行处理
6.  appArgs.action match {
7.        case SparkSubmitAction.SUBMIT => submit(appArgs)
8.        case SparkSubmitAction.KILL => kill(appArgs)
9.        case SparkSubmitAction.REQUEST_STATUS => requestStatus(appArgs)
10.     }
11.  }
```

其中，第 3 行代码中的 SparkSubmitArguments 类对应用户调用提交脚本 spark - submit 时传入的参数信息。对应的脚本的帮助信息（./bin/spark - submit -- help）也是由该类的 printUsageAndExit 方法提供的。

找到上面的入口点代码之后，就可以开始分析其内部的源代码。对应参数信息的 SparkSubmitArguments 可以参考脚本的帮助信息来查看具体参数所对应的含义。参数分析之后，

便是各种提交行为的具体处理。SparkSubmit 支持 SparkSubmitAction 包含的 3 种行为，下面以行为 SparkSubmitAction. SUBMIT 为例进行分析，其他行为也可以从各自的具体处理代码进行分析。

对应处理 SparkSubmitAction. SUBMIT 行为的代码入口点为 submit(appArgs)，进入该方法，即进入提交应用程序的处理方法的具体代码如下。

```
// SparkSubmitAction. SUBMIT 行为类型的具体处理方法
private def submit(args:SparkSubmitArguments):Unit = {
// 准备应用程序提交的环境,该步骤包含了内部封装的各个细节处理
  val(childArgs,childClasspath,sysProps,childMainClass) = prepareSubmitEnvironment(args)

    def doRunMain():Unit = {
      if (args. proxyUser!= null) {
        val proxyUser = UserGroupInformation. createProxyUser(args. proxyUser,
          UserGroupInformation. getCurrentUser())
        try {
          proxyUser. doAs(new PrivilegedExceptionAction[Unit]() {
            override def run():Unit = {
              runMain(childArgs,childClasspath,sysProps,childMainClass,args. verbose)
            }
          })
        } catch {
 …
        }
      } else {
        runMain(childArgs,childClasspath,sysProps,childMainClass,args. verbose)
      }
    }

    //  Standalone 集群模式下,有两种提交应用程序的方式

//1、传统的 Akka 网关方式使用 o. a. s. deploy. Client 进行封装
//2、Spark 1. 3 使用了 REST - based 网关方式,作为 Spark 1. 3 的缺省方法,如果 master 结点
不是 REST 服务器结点,spark 应用程序提交时会切换到传统的网关模式
    if (args. isStandaloneCluster && args. useRest) {
      try {
        // scalastyle:off println
          printStream. println ( " Running Spark using the REST application submission
          protocol. " )
        // scalastyle:on println
        doRunMain()
      } catch {
```

```
            // 失败的话则使用传统的提交方式

            case e:SubmitRestConnectionException =>
              printWarning(s"Master endpoint${args.master} was not a REST server.   " +
                "Falling back to legacy submission gateway instead.")
// 重新设置提交方式的控制开关
args.useRest = false
              submit(args)
          }
        // In all other modes,just run the main class as prepared
        } else {
          doRunMain()
        }
    }
```

其中，最终运行所需的参数都由 prepareSubmitEnvironment 方法负责解析和转换，然后根据其结果执行。解析的结果包含以下 4 部分。

- 子进程运行所需的参数。
- 子进程运行时的 classpath 列表。
- 系统属性的映射。
- 子进程运行时的主类。

解析之后调用 runMain 方法，该方法中除了一些环境设置等操作外，最终会调用解析得到的 childMainClass 的 main 方法。下面简单分析一下 prepareSubmitEnvironment 方法，通过该方法来了解 SparkSubmit 是如何帮助底层的集群管理器和部署模式的封装。接下来以不同集群管理器和部署模式下最终运行的 childMainClass 类的解析为主线进行分析。

1）当部署模式为 Client 时，将 childMainClass 设置为传入的 mainClass，对应代码如下。

```
// 在 client 模式,直接启动应用程序的主类
// 同时,将主类的 jar 包和添加的 jars 包(如果在参数中设置的话)
// 都添加到运行时的 classpath 中
if(deployMode == CLIENT) {
  childMainClass = args.mainClass
  if(isUserJar(args.primaryResource)) {
    childClasspath += args.primaryResource
  }
  if(args.jars != null) { childClasspath ++= args.jars.split(",") }
   if(args.childArgs != null) { childArgs ++= args.childArgs }
}
```

2）当集群管理器为 Standalone、部署模式为 Cluster 时，根据提交的两种方式将 childMainClass 分别设置为不同的类，同时将传入的 args.mainClass（提交应用程序时所设置的主

类）及其参数根据不同集群管理器与部署模式进行转换，并封装到新的主类所需的参数中。对应的设置如表 3-3 所示。

表 3-3 Standalone + Cluster 时两种不同提交方式下的 childMainClass 封装

提 交 方 式	childMainClass
REST 方式（Spark 1. 3 + ）	" org. apache. spark. deploy. rest. RestSubmissionClient"
传统方式	" org. apache. spark. deploy. Client"

表 3-3 中，REST（representational State Transfer，表述性状态传递）是 Roy Fielding 博士在 2000 年他的论文中提出来的一种软件架构网格。

这些设置的主类相当于封装了应用程序提交时的主类，运行后负责向 Master 结点申请启动提交的应用程序。

3）当集群管理器为 YARN、部署模式为 Cluster 时，childMainClass 及对应的 mainClass 的设置如表 3-4 所示。

表 3-4 YARN + Cluster 时 childMainClass 下的 childMainClass 封装

执 行 对 象	childMainClass	被封装的执行类（mainClass）
isPython	" org. apache. spark. deploy. yarn. Client"	" org. apache. spark. deploy. PythonRunner"
isR		" org. apache. spark. deploy. RRunner"
其他		args. mainClass

4）当集群管理器为 Mesos、部署模式为 Cluster 时，childMainClass 及对应的 mainClass 的设置如表 3-5 所示。

表 3-5 Mesos + Cluster 时 childMainClass 下的 childMainClass 封装

执 行 对象	childMainClass	被封装的执行类（mainClass）
isPython	" org. apache. spark. deploy. rest. RestSubmissionClient"	" "
其他		args. mainClass

从上面的分析可以看到，当使用 Clinet 部署模式进行提交时，由于设置的 childMainClass 为应用程序提交时的主类，因此是直接在提交点执行设置的主类，即 mainClass。当使用 Cluster 部署模式进行提交时，则会根据具体集群管理器等信息使用相应的封装类。这些封装类会向集群申请提交应用程序的请求，然后在由集群调度分配得到的结点上启动所申请的应用程序。

以将封装类设置为" org. apache. spark. deploy. Client" 为例，从该类主入口 main 方法查看，可以看到构建了一个 ClientEndpoint 实例，在构建该实例时，会将提交应用程序时设置的 mainClass 等信息封装到 DriverDescription 实例中，然后发送到 Master，申请执行用户提交的应用程序。

对应各种集群管理器与部署模式的组合，实际代码中的处理细节非常多。这里仅仅给出了一种源代码阅读的方式，和对应的大数据处理一样，通常采用化繁为简的方式去阅读复杂

的源代码。比如这里在理解了整个大框架的调用过程后，以 childMainClass 的设置作为主线去解读源代码，相应的，在扩展阅读其他源代码时，也可以采用这种方式，以某种集群管理器与部署模式为主线，详细阅读相关的代码。最后，在了解各种组合的处理细节之后，通过对比、抽象等方法，对整个 SparkSubmit 进行归纳总结即可。

参考 3.4.1 部署框架一章中的集群部署组件图（图 3-2），可以看到提交的应用程序的驱动程序（Driver Program）部分对应包含了一个 SparkContext 实例。因此接下来从该实例出发，解析驱动程序在不同的集群管理器的部署细节。

4. SparkContext 解析

在详细解析 SparkContext 实例之前，首先来看下 SparkContext 类的注释部分，具体如下。

```
/**
  * Spark 功能的主入口点
 * Main entry point for Spark functionality. A SparkContext represents the connection to a
  * Spark cluster,and can be used to create RDDs,accumulators and broadcast variables on
 * that cluster.
  * …
  * @param config a Spark Config object describing the application configuration. Any
 * settings inthis config overrides the default configs as well as system properties.
  */
```

SparkContext 类是 Spark 功能的主入口点。一个 SparkContext 实例代表了与一个 Spark 集群的连接，并且通过该实例可以在集群中构建 RDD、累加器及广播变量。SparkContext 实例的构建参数 config，描述了应用程序的 Spark 配置。在该参数中指定的配置属性会覆盖默认的配置属性及系统属性。

在 SparkContext 类文件中，定义了一个描述集群管理器类型的单例对象 SparkMasterRegex，在该对象中详细给出了当前 Spark 所支持的各种集群管理器类型，具体代码如下。

```
/**
  * 定义了从 master 信息中抽取集群管理器类型的一个正则表达式集合
  * A collection of regexes for extracting information from the master string.
  */
private object SparkMasterRegex {
// 对应 master 格式如 local[N] 和 local[*]的正则表达式
 // Regular expression used for local[N] and local[*] master formats
 val LOCAL_N_REGEX = """local\[([0-9]+|\*)\]""".r

 // 对应 master 格式如 local[N,maxRetries]的正则表达式
  // 这种集群管理器类型用于具有任务失败尝试功能的测试
 // Regular expression for local[N,maxRetries],used in tests with failing tasks
   val LOCAL_N_FAILURES_REGEX = """local\[([0-9]+|\*)\s*,\s*([0-9]+)
   \]""".r

```

```
// 一种模拟 Spark 集群的本地模式的正则表达式
// 对应的 master 格式如 local-cluster[N,cores,memory]
// Regular expression for simulating a Spark cluster of [N,cores,memory] locally
  val LOCAL_CLUSTER_REGEX = """local-cluster\[\s*([0-9]+)\s*,\s*([0-9]
  +)\s*,\s*([0-9]+)\s*]""".r

// 连接 Spark 部署集群的正则表达式
 // Regular expression for connecting to Spark deploy clusters
  val SPARK_REGEX = """spark://(.*)""".r

// 连接 Mesos 集群的正则表达式
// Regular expression for connection to Mesos cluster by mesos:// or mesos://zk:// url
  val MESOS_REGEX = """mesos://(.*)""".r

// 使用 Spark In MapReduce 方式的正则表达式。在 MapReduce 1 中使用 Spark 时用
 // Regular expression for connection to Simr cluster
  val SIMR_REGEX = """simr://(.*)""".r
}
```

在 SparkContext 类中的主要流程可以归纳如下。

1）createSparkEnv：创建 Spark 的执行环境对应的 SparkEnv 实例。

对应代码如下。

```
// Create the Spark execution environment(cache,map output tracker,etc)
_env = createSparkEnv(_conf,isLocal,listenerBus)
SparkEnv.set(_env)
```

2）createTaskScheduler：创建作业调度器实例。

对应代码如下。

```
// Create and start the scheduler
val (sched,ts) = SparkContext.createTaskScheduler(this,master)
_schedulerBackend = sched
_taskScheduler = ts
```

其中，TaskScheduler 是低层次的任务调度器，负责任务的调度。通过该接口提供可插拔的任务调度器。每个 TaskScheduler 负责调度一个 SparkContext 实例中的任务，负责调度上层 DAG 调度器中每个 Stage 提交的任务集（TaskSet），并将这些任务提交到集群中运行，在任务提交执行时可以使用失败重试机制设置失败重试的次数。上述对应高层的 DAG 调度器的实例构建请参见下一步。

3）newDAGScheduler：创建高层 Stage 调度的 DAG 调度器实例。

对应代码如下。

```
_dagScheduler = new DAGScheduler(this)
```

DAGScheduler 是高层调度模块，负责作业（Job）的 Stage 拆分，以及最终将 Stage 对应的任务集提交到低层次的任务调度器上。

下面基于这些主要流程，针对 SparkMasterRegex 单例对象中给出的各种集群部署模式进行解析。对应不同集群模式下，这些流程中构建了包括 TaskScheduler 与 SchedulerBackend 的不同的具体子类，所构建的相关实例具体如表 3-6 所示。

表 3-6　各种情况下 TaskScheduler 与 SchedulerBackend 的不同的具体子类

部署模式	实例对应的类	备　注
Local	_taskScheduler:TaskSchedulerImpl _schedulerBackend:LocalBackend	最简单的本地模式 这种本地模式下，任务的失败重试次数为 1，即失败不重试
Local[*]、Local[N]		指定线程个数的本地模式，指定方式及最终的线程数如下： 1）Local[*]：当前处理器的个数 2）Local[N]：指定的 N 这种本地模式下，任务的失败重试次数为 1，即失败不重试
Local[* ,M]、Local[N,M]		指定线程个数及失败重试次数的本地模式，仅比上一种本地模式多了一个失败重试次数的设置，对应为 M
Local - Cluster [numSlaves, coresPerSlave, memoryPerSlave]	_taskScheduler:TaskSchedulerImpl _schedulerBackend:SparkDeploySchedulerBackend	本地伪分布式集群，由于本地模式下没有集群，因此需要构建一个用于模拟集群的实例： localCluster = new LocalSparkCluster 对应的 3 个参数如下： numSlaves：模拟集群的 Slave 结点个数 coresPerSlave：模拟集群的各个 Slave 结点上的内核数 memoryPerSlave：模拟集群的各个 Slave 结点上的内存大小
Spark Standalone		Spark Standalone 对应 Spark 原生的完全分布式集群 因此，此种方式下不需要像上面的本地伪分布式集群那样构建一个虚拟的本地集群
YARN Client	_taskScheduler:YarnScheduler _schedulerBackend:YarnClientSchedulerBackend	YARN 集群管理器 + Client 部署
YARN Cluster	_ taskScheduler:YarnClusterScheduler _schedulerBackend:YarnClusterSchedulerBackend	YARN 集群管理器 + Cluster 部署
Mesos	_taskScheduler:YarnScheduler _schedulerBackend:CoarseMesosSchedulerBackend、MesosSchedulerBackend	_schedulerBackend 具体的实例根据资源分配的请求设置
Simr	_taskScheduler:TaskSchedulerImpl _schedulerBackend:LocalBackend	Spark InMapReduce 1 的集群部署

与 TaskScheduler 与 SchedulerBackend 不同的是，在不同集群模式中，应用程序的高层调度器 DAGScheduler 的实例是相同的，即对应在 Spark on YARN 与 Mesos 等集群管理器中，应用程序内部的高层 Stage 调度是相同的。

Section 3.3 Local 与 Local – Cluster 部署

对于分布式集群，通常都会提供一种方便初学者入门学习和测试的部署模式，也就是在 Hadoop 框架中常说的 Local（本地）模式。同样的，Spark 框架也不例外，而且相对于其他框架，比如 Hadoop，Spark 框架提供了更丰富的 Local 模式。

在进一步解析之前，首先来看下 SparkSubmit 提交应用程序时，一些不支持的组合形式，对应代码如下。

```
private[deploy] def prepareSubmitEnvironment(args:SparkSubmitArguments)
…
case(LOCAL,CLUSTER) =>
printErrorAndExit("Cluster deploy mode is not compatible with master\"local\"")
…
```

即，在使用 Local 与 Local – Cluster 这两种 Local 方式时，不支持以 CLUSTER 的部署模式提交应用程序。

了解 SparkContext 执行的主要流程后，再以 Local 与 Local – Cluster 部署为主线，分析本地集群部署模式下的处理细节。

可以从前面的分析中抽取出本地模式部署的 TaskScheduler 与 SchedulerBackend 的具体子类的实例构建信息，如表 3–7 所示。

表 3–7　Local 模式部署时的 TaskScheduler 与 SchedulerBackend 的具体子类

<table>
<tr><th>部署模式（Master）</th><th>实例对应的类</th><th>备　注</th></tr>
<tr><td>Local</td><td rowspan="3">_taskScheduler:TaskSchedulerImpl
_schedulerBackend:LocalBackend</td><td>最简单的本地模式
这种本地模式下，任务的失败重试次数为 1，即失败不重试</td></tr>
<tr><td>Local[*]、Local[N]</td><td>指定线程个数的本地模式，指定方式及最终的线程数如下：
1）Local[*]：当前处理器的个数
2）Local[N]：指定的 N
这种本地模式下，任务的失败重试次数为 1，即失败不重试</td></tr>
<tr><td>Local[* ,M]、
Local[N,M]</td><td>指定线程个数及失败重试次数的本地模式，仅比上一种本地模式多了一个失败重试次数的设置，对应为 M</td></tr>
<tr><td>Local – Cluster[numSlaves,coresPerSlave,memoryPerSlave]</td><td>_taskScheduler:TaskSchedulerImpl
_schedulerBackend:SparkDeploySchedulerBackend</td><td>本地伪分布式集群，由于本地模式下没有集群，因此需要构建一个用于模拟集群的实例：localCluster = new LocalSparkCluster
对应的 3 个参数如下：
numSlaves：模拟集群的 Slave 结点个数
coresPerSlave：模拟集群的各个 Slave 结点上的内核数
memoryPerSlave：模拟集群的各个 Slave 结点上的内存大小</td></tr>
</table>

本地部署模式可以分为本地模式和本地伪分布式模式，其中，本地模式又分 3 种，而 Local－Cluster 这种则对应本地伪分布式模式。在 3 种本地模式中，内部实现的实例相同，仅仅是启动的线程数与任务失败重试的次数不同；本地伪分布式模式中，通过提供与 Spark Standalone 部署集群对应的信息来模拟完全分布式的集群。

下面分别针对这几种本地部署模式进行分析。

3.3.1 Local 部署

在该模式下，使用一个工作线程执行计算任务，并且在任务失败时不会重新计算，即在失败重启机制中使用重启次数为 1。对应的控制变量参考代码如下。

```
// 当本地运行时,任务失败时不会重新执行
// When running locally,don't try to re-execute tasks on failure
val MAX_LOCAL_TASK_FAILURES = 1
```

3.3.2 Local[*]与 Local[N]部署

该模式相当于在 Local 部署的基础上增加了线程个数的控制。

1）Local[*]模式，线程个数取决于本机的处理器个数，逻辑上保证一个处理器对应一个处理线程。

2）Local[N]模式，直接指定了使用的线程个数为 N。

在这两种模式下，也不会启动任务失败重试的机制，对应控制参数同 Local 模式。

3.3.3 Local[*,M]与 Local[N,M]部署

在该模式下，除了和上一种类似的线程个数控制之外，还增加了任务失败重试机制的失败重启次数的配置，即可以指定任务失败重试的最大次数为 M。

在前 3 种本地模式中，SchedulerBackend 的实现都是 LocalBackend，该类的注释如下。

```
/**
  * 在以下情况时使用 LocalBackend:运行一个 Spark 的本地模式,其中,
  * executor、backend 和 master 运行在同一个 JVM 进程中
 * LocalBackend is used when running a local version of Spark where the executor,backend,
  * and master all run in the same JVM. It sits behind a TaskSchedulerImpl and handles
  * launching tasks on a single Executor(created by the LocalBackend)running locally.
  */
private[spark] class LocalBackend(
    conf:SparkConf,
      scheduler:TaskSchedulerImpl,
      val totalCores:Int)
```

在此简单给出内部实现流程的解析，具体步骤如下。

1）对应的初始化代码在前面提到的 SparkContext 类中主要流程的 createTaskScheduler 方法中，构建 TaskScheduler 实例（这里具体子类为 TaskSchedulerImpl）后，在初始化该实例时传入同时构建的 SchedulerBackend 实例（这里具体子类为 LocalBackend）。

2）构建出 TaskScheduler 实例后，会调用实例的 start 方法，在该方法中首先会调用 SchedulerBackend 的 start 方法。

3）在 SchedulerBackend 的 start 方法中会构建出一个 LocalEndpoint 实例，在该实例中就会实例化出一个 Executor，Executor 实例负责具体的任务执行。

4）之后就是 TaskScheduler 进行作业调度，调用 SchedulerBackend 的 reviveOffers( ) 方法，然后由该方法向 LocalEndpoint 实例发送 ReviveOffers 消息。

5）最终在 LocalEndpoint 实例处理 ReviveOffers 消息时启动 Task，其他处理类似。对应 Task 的启动代码如下。

```
def reviveOffers( ) {
  val offers = Seq( new WorkerOffer( localExecutorId,localExecutorHostname,freeCores) )
  for( task <- scheduler. resourceOffers( offers). flatten) {
    // 在 Executor 中会使用线程池的方式调度任务,而对应的作业调度是通过
    // 判断当前可用 Cores 个数是否符合每个任务(Task)所需的 Cores 个数。
    // 当符合该条件时更新当前可用 Cores 数 freeCores,然后启动任务(Task)
    freeCores -= scheduler. CPUS_PER_TASK
        executor. launchTask ( executorBackend, taskId = task. taskId, attemptNumber =
  task. attemptNumber,
      task. name,task. serializedTask)
   }
 }
```

其中，Task 的调度控制代码参考 TaskSchedulerImpl 的 resourceOfferSingleTaskSet 方法。

上述 3 种 Local 的部署模式可以通过图 3-1 来加深理解。

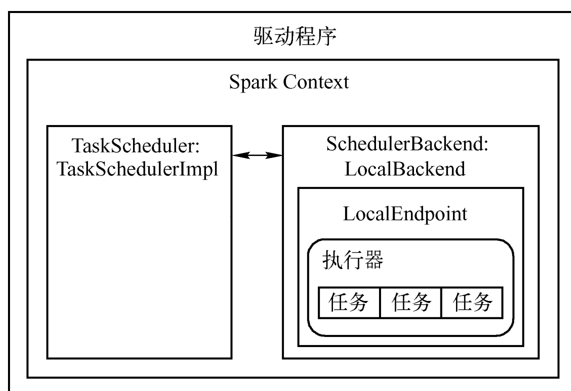


图 3-1　3 种 Local 的部署模式图

其中，TaskScheduler 与 SchedulerBackend 的具体子类的具体子类分别为 TaskSchedulerImpl 与 LocalBackend，具体的 Task 仍然在 Executor 中执行。

3.3.4 Local – Cluster[S,C,M]部署

格式如 Local – Cluster ［numSlaves，coresPerSlave，memoryPerSlave］ 的这种模式称为本地伪分布式部署模式，由于当前使用的是本地部署模式，因此不存在所谓的集群，所以在模拟伪分布式部署模式时，需要构建出一个模拟的集群模式。模拟的集群模式在代码中对应 LocalSparkCluster 实例。

在本地伪分布式部署模式中，构建的作业调度器同其他 3 种本地模式一样，也是实例化具体子类 TaskSchedulerImpl，但同时构建的 SchedulerBackend 实例是和真实的 Spark Standalone 集群是一样的，也是实例化了 SparkDeploySchedulerBackend 子类。这说明本地伪分布式部署模式仅仅在集群组件构建的方式上有所差异，其他方面都是相同的。

对应的集群模拟可以查看 LocalSparkCluster 的 start 方法，其中构建了 Master 和多个 Worker 实例来模拟分布式集群。模拟时使用的参数也是参考 Spark Standalone 集群，通过 numSlaves 指定模拟集群中的 Slaves 结点个数，通过 coresPerSlave 指定模拟集群中各个 Slave 结点上的内核数，以及通过 memoryPerSlave 指定模拟集群中各个 Slave 配置的内存大小。

另外，本地模式在其他细节方面的影响，可以查看 SparkContext 中的本地模式控制变量的设置，对应变量定义的代码如下。

```
def isLocal:Boolean = (master == "local" || master.startsWith("local["))
```

通过查看 isLocal 变量的代码，即可找到与本地模式相关的内容。

Section 3.4 Spark Standalone 部署

3.4.1 部署框架

在详细解析之前，首先来看一下集群部署组件图，如图 3–2 所示。

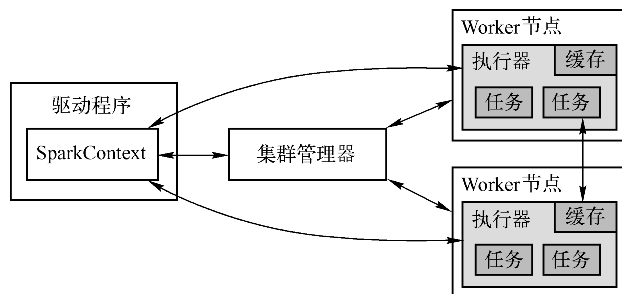


图 3–2　集群部署组件图

其中各个术语及相关术语的描述如下。

1）Driver Program：运行 Application 的 main 函数并且新建 SparkContext 实例的程序，称为驱动程序（Driver Program）。通常可以使用 SparkContext 来代表驱动程序（Driver Program）。

2）Cluster Manager：集群管理器（Cluster Manager），是集群资源管理的外部服务，在 Spark 上现在主要有 Standalone、YARN 和 Mesos 等 3 种集群资源管理器，Spark 自带的 Standalone 模式能够满足绝大部分纯粹的 Spark 计算环境中对集群资源管理的需求，基本上只有在集群中运行多套计算框架时才建议考虑 YARN 和 Mesos。

3）Worker Node：集群中可以运行 Application 代码的工作结点（Worker Node），相当于 Hadoop 的 Slave 结点。

4）Executor：在 Worker Node 上为 Application 启动的一个工作进程，在进程中负责任务（Task）的运行，并且负责将数据存放在内存或磁盘上，在 Executor 内部通过多线程的方式（即线程池）并发处理应用程序的具体任务。

每个 Application 都有各自独立的 Executors，因此应用程序之间是相互隔离的。

5）Task：任务（Task）是指被 Driver 送到 Executor 上的工作单元，通常情况下一个任务（Task）会处理一个 Partition 的数据，每个 Partition 一般为一个 HDFS 的 Block 块的大小。

6）Application：Application 是创建了 SparkContext 实例对象的 Spark 用户程序，包含了一个 Driver program 和集群中多个 Worker 上的 Executor。

7）Job：和 Spark 的 action 相对应，每一个 action，如 count、savaAsTextFile 等都会对应一个 Job 实例，每个 Job 会被拆分成多个 Stages，一个 Stage 中包含一个任务集（TaskSet），任务集中的各个任务（Task）通过一定的调度机制发送到工作单位（Executor）上并行执行。

3.4.2 应用程序的部署

和其他常见的分布式集群类似，Spark Standalone 集群的部署也是采用典型的 Master/Slave 架构。其中，Master 结点负责整个集群的资源管理与调度，Worker 结点（也可以称 Slave 结点）在 Master 结点的调度下启动 Executor，负责执行具体工作（包括应用程序及应用程序提交的任务）。

从前面的分析中抽取出 Spark Standalone 模式部署的 TaskScheduler 与 SchedulerBackend 具体子类的实例构建信息，如表 3-8 所示。

表 3-8 Spark Standalone 模式部署具体子类的构建

部署模式（master）	实例对应的类	备 注
Spark Standalone	taskScheduler:TaskSchedulerImpl _ schedulerBackend：SparkDeployScheduler-Backend	Spark Standalone 对应 Spark 原生的完全分布式集群。 因此，此种方式下不需要像上面的本地伪分布式集群那样构建一个虚拟的本地集群

下面以提交请求的行为为例，结合应用程序提交时所使用的不同部署模式，给出详细的框架及其描述，对应在框架中的其他请求与此类似，可以自行解析。

1. 以 Client 的部署模式提交应用程序

在 Client 的部署模式提交时，直接在提交点运行应用程序，即对应的驱动程序是在当前结点启动的。启动一个应用程序后，就涉及各个相关的方面，包含应用运行的环境、应用元数据的清理、状态监听、DAG 调度、任务调度等。

对应的部署与执行框架如图 3-3 所示。

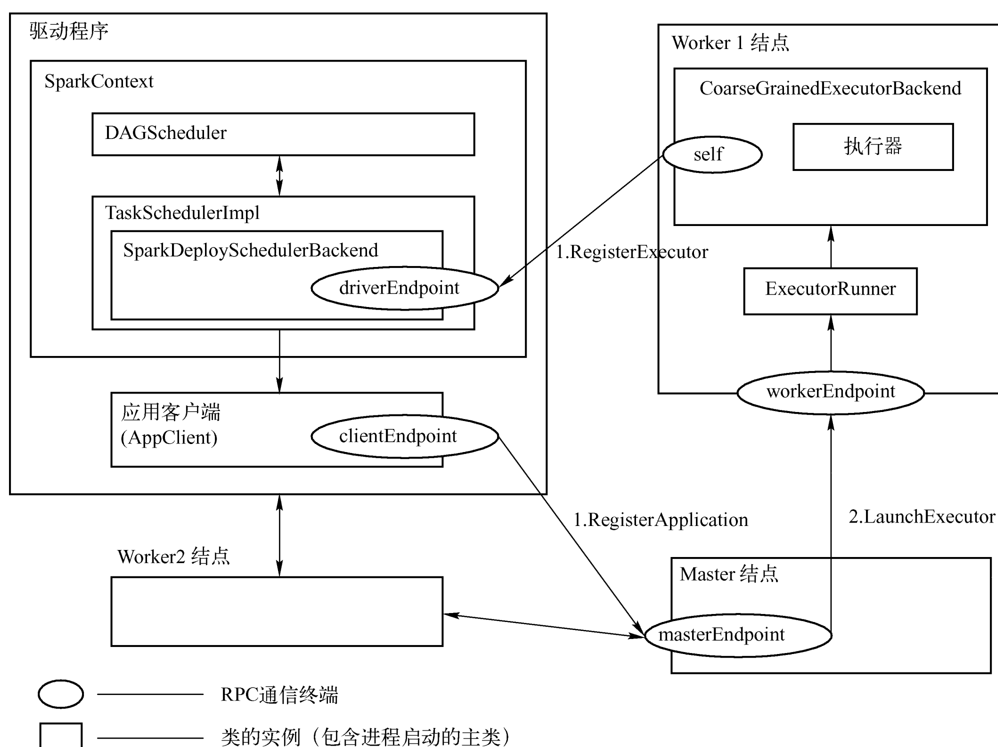


图 3-3　Client 部署模式下的部署与执行框架

如图 3-3 所示，在驱动程序（Driver Program）内部会构建一个 SparkContext 实例，在前面章节中已经分析过，在 SparkContext 实例的主要流程中，会构建出用于任务调度的 TaskScheduler 实例、用于 DAG 调度的 DAGScheduler 实例，以及作为 TaskSchedulerImpl 底层的一个可插拔的调度系统终端的 SparkDeploySchedulerBackend 实例。由于调度系统后续会用单独一章的篇幅进行解析，因此这里仅仅给出简单的部署与交互过程，具体过程如下。

1）在 SparkContext 构建出 SparkDeploySchedulerBackend 实例后，调用该实例的 start 方法，关键代码如下。

```
override def start( ) {
    super. start( )
```



```
    launcherBackend. connect( )
  …
    // 封装的命令,该命令发送到 Worker 结点,并根据获取的资源启动后,相当于
    // 打开了一个通信通道
  val command = Command( "org. apache. spark. executor. CoarseGrainedExecutorBackend" ,
          args, sc. executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)
      val appUIAddress = sc. ui. map( _. appUIAddress). getOrElse( "" )
        val coresPerExecutor = conf. getOption( "spark. executor. cores" ). map( _. toInt)
   // 通过 ApplicationDescription 将 command 封装起来
   val appDesc = new ApplicationDescription( sc. appName, maxCores, sc. executorMemory,
            command, appUIAddress, sc. eventLogDir, sc. eventLogCodec, coresPerExecutor)

   // 构建一个作为应用程序客户端的 AppClient 实例,并将 this 设置为该实例的监听
   // 器,AppClient 实例内部会将 Executor 端的消息转发给 this
    client = new AppClient( sc. env. rpcEnv, masters, appDesc, this, conf)
        client. start( )
        launcherBackend. setState( SparkAppHandle. State. SUBMITTED)
        waitForRegistration( )
        launcherBackend. setState( SparkAppHandle. State. RUNNING)
    }
```

其中，AppClient 实例在调用方法 start 时，会构建一个 RPC 通信终端，即 ClientEndpoint 实例，实例化后再自动调用 onStart( )，这时就会将封装的 ApplicationDescription 实例进一步封装到消息 RegisterApplication 的实例中，然后由该 RPC 通信终端将该信息发送到 Master 的 RPC 通信终端。

2）Master 的 RPC 通信终端在收到 RegisterApplications 消息后，通过资源调度方法，最终会调用 launchExecutor 方法，在该方法中再向调度所分配到的 Worker 结点的 RPC 通信终端发送 LaunchExecutor 消息。

3）Worker 的 RPC 通信终端在收到 LaunchExecutor 消息后，会实例化 ExecutorRunner 对象，然后启动一个线程，在线程中解析 RegisterApplications 消息封装的 ApplicationDescription 实例所携带的 Command 实例，也就是前面封装的 CoarseGrainedExecutorBackend 类，最后启动 CoarseGrainedExecutorBackend 类的进程。进程的入口就是 CoarseGrainedExecutorBackend 伴生对象的 main 函数。

4）在入口处，即 CoarseGrainedExecutorBackend 伴生对象的 main 函数中，会解析参数，然后调用 run 函数，在该 run 函数中会构建 CoarseGrainedExecutorBackend 实例，也就是构建一个 RPC 通信终端。Run 方法中的关键代码如下。

```
env. rpcEnv. setupEndpoint( "Executor" , new CoarseGrainedExecutorBackend(
  env. rpcEnv, driverUrl, executorId, sparkHostPort, cores, userClassPath, env) )
workerUrl. foreach { url =>
```

```
    env.rpcEnv.setupEndpoint("WorkerWatcher",new WorkerWatcher(env.rpcEnv,url))
}
```

其中，driverUrl 是封装 CoarseGrainedExecutorBackend 到 Command 时设置的，可以回到前面 SparkDeploySchedulerBackend 实例的 start 方法，在构建 Command 之前，设置了一些参数，对应的代码如下。

```
1.      // The endpoint for executors to talk to us
2.      val driverUrl = rpcEnv.uriOf(SparkEnv.driverActorSystemName,
3.          RpcAddress(sc.conf.get("spark.driver.host"), sc.conf.get("spark.driver.port")
   .toInt),
4.  CoarseGrainedSchedulerBackend.ENDPOINT_NAME)
5.      val args = Seq(
6.   "--driver-url",driverUrl,
7.        "--executor-id","{{EXECUTOR_ID}}",
8.        "--hostname","{{HOSTNAME}}",
9.        "--cores","{{CORES}}",
10.        "--app-id","{{APP_ID}}",
11.        "--worker-url","{{WORKER_URL}}")
```

其中，第 4、6 行就是封装的与 CoarseGrainedExecutorBackend 进行通信的终端及其对应参数的选项名称，也就是前面的 CoarseGrainedSchedulerBackend 实例的 driverEndpoint：DriverEndpoint 成员，对应在 Spark Standalone 部署模式下，就是具体子类 SparkDeploySchedulerBackend。

5）对应 CoarseGrainedExecutorBackend 的 RPC 通信终端，在实例化时自动调用 onStart 方法。在该方法中向 driverUrl 发送 RegisterExecutor 消息。这里的 driverUrl 就是上一步分析得到的驱动程序端的 SparkDeploySchedulerBackend 的 RPC 通信端口 driverEndpoint。

6）SparkDeploySchedulerBackend 实例收到 RegisterExecutor 消息时，表示当前有可用资源注册上来，此时即可开始作业的调度。具体的调度过程可以参考本书的第 4 章。

说明：分布式消息间的调度是建立在消息事件的基础上的，可以通过列举所有的 RPC 通信终端（系统名 + 终端名）和各个终端之间交互的信息这两个方面，来理清整个分布式集群中各个组件之间和内部的交互逻辑，进而把握整个调度机制。

2. 以 Cluster 的部署模式提交应用程序

结合“3.2.2 应用部署的源代码解析”中的内容来看，在 Cluster 的部署模式提交时，Spark 会将应用程序封装到指定的类中，由该类负责向集群申请提交应用程序的执行。即在 Cluster 部署模式提交时，通过向 Master 申请执行应用程序，然后由 Master 负责调度分配一个 Worker 结点，并向该结点发送启动应用程序的消息，应用程序启动后的执行流程，与在该 Worker 结点上直接以 Client 部署模式提交应用程序的执行流程是一样的，只是在此之前

需要先调度得到该 Worker 结点，并在该结点启动应用程序。由于在申请到资源并启动提交的应用程序之后，这一执行框架和 Client 部署模式下是一样的，因而在此仅关注将应用程序封装到指定类，并与集群 Master 进行交互的执行框架。

再次查看对应封装的类，如表 3-9 所示。

表 3-9 Cluster 的部署模式提交时具体封装的子类

提交方式	childMainClass
REST 方式（Spark 1.3+）	"org.apache.spark.deploy.rest.RestSubmissionClient"
传统方式	"org.apache.spark.deploy.Client"

下面根据不同的提交方式分析提交应用程序的请求，对应其他的请求行为，比如 Kill 应用程序或获取应用程序的状态等，都可以借鉴提交应用的执行框架的解析来加深理解。

说明：注意 Client 与 Cluster 两种部署模式下，首次提交申请的对象是不同的，Clinet 时是直接提交应用程序的申请，Cluster 时会先提交应用程序本身的申请，结合前面的部署框架（见图 3-2），通常应用程序也称为驱动程序（Driver Program），因此该部署模式下首次提交的申请对应的是 Driver 的申请，在代码中这两种模式分别对应的申请的描述信息为 ApplicationDescription 与 DriverDescription。

（1）提交方式为 REST 方式（Spark 1.3+）

当提交方式为 REST 方式（Spark 1.3+）时，Spark 会将应用程序的主类等信息封装到 RestSubmissionClient 类中，由该类负责向 RestSubmissionServer 发送提交应用程序的请求，而 RestSubmissionServer 接收到应用程序提交的请求之后，会向 Master 发送 RequestSubmitDriver 消息，然后由 Master 根据资源调度策略，启动集群中相应的 Driver，执行提交的应用程序。详细的执行框架如图 3-4 所示。

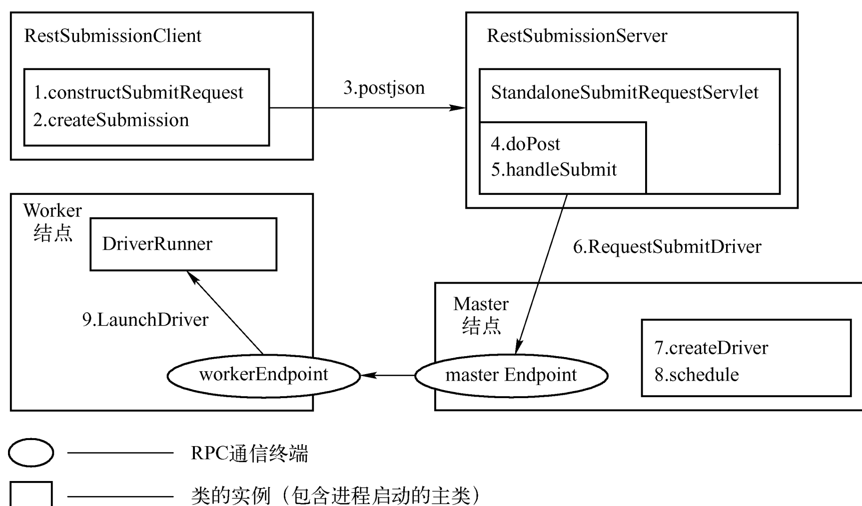


图 3-4 Cluster 部署模式下的部署与执行框架

为了体现各个组件间的部署关系，这里以框架图的形式进行描述，相应的，可以从时序图的角度去理解各个类或组件之间的交互关系。其中组件 Master 和 Worker 的标注在方框的

左上角，其他方框表示一个具体的实例。

其中，RestSubmissionClient 是提交应用程序的客户端处，是对提交的应用程序进行封装的类。之后各个组件间的交互流程分析如下。

1）第 1 步 constructSubmitRequest，就是在 RestSubmissionClient 实例中，根据提交的应用程序信息构建出提交请求。

2）然后继续第 2 步 createSubmission，在该步骤中向 RestSubmissionServer 发送 post 请求，即图 3-4 中对应的第 3 步（注意，实际上是在第 2 步中调用）。

3）RestSubmissionServer 接收到 post 请求后，由对应的 Servlet 进行处理，这里对应为 StandaloneSubmitRequestServlet。即开始第 4 步，调用 doPost，发送 Post 请求。

4）doPost 中继续第 5 步 handleSubmit，开始处理提交请求。在处理过程中，向 Master 的 RPC 终端发送消息 RequestSubmitDriver，对应图 3-4 中的第 6 步。

5）Master 接收到该消息后，执行第 7 步 createDriver，创建 Driver，创建时需要由 Master 的调度机制（对应第 8 步 schedule）获取分配的资源后，向 Worker（这些 Worker 启动时会注册到 Master 上）的 RPC 终端发送 LaunchDriver 消息。

6）Worker 在 RPC 终端接收到消息后开始处理并实例化一个 DriverRunner，并运行之前封装的应用程序。

注意：从上面的部署框架及其术语解析部分可以知道，由于提交的应用程序在 main 部分包含了 SparkContext 实例，因此也称之为 Driver Program，即驱动程序。因此在框架中，对应在 Master 和 Worker 处都使用 Driver，而不是 Application（应用程序）。

其中主要的源代码及其分析如下。

1）RestSubmissionClient 的 run 方法的代码如下。

```
/**
   * Submit an application, assuming Spark parameters are specified through the given config.
   * This is abstracted to its own method for testing purposes.
   */
  def run(
      appResource: String,
      mainClass: String,
      appArgs: Array[String],
      conf: SparkConf,
        env: Map[String, String] = Map()): SubmitRestProtocolResponse = {
      val master = conf.getOption("spark.master").getOrElse {
        throw new IllegalArgumentException("spark.master must be set.")
      }
      val sparkProperties = conf.getAll.toMap

 //创建一个 Rest 提交客户端
      val client = new RestSubmissionClient(master)
```

```
// 封装应用程序的相关信息,包括主资源、主类等
    val submitRequest = client.constructSubmitRequest(
      appResource,mainClass,appArgs,sparkProperties,env)

    // Rest 提交客户端开始创建 Submission,
    // 创建过程中向 RestSubmissionServer 发送 post 请求
    client.createSubmission(submitRequest)
  }
```

2）收到提交的 Post 请求之后，StandaloneSubmitRequestServlet 向 Master 的 RPC 终端发送请求，代码如下。

```
  /**
   * Handle the submit request and construct an appropriate response to return to the client.
   *
   * This assumes that the request message is already successfully validated.
   * If the request message is not of the expected type,return error to the client.
   */
  protected override def handleSubmit(
      requestMessageJson:String,
      requestMessage:SubmitRestProtocolMessage,
       responseServlet:HttpServletResponse):SubmitRestProtocolResponse = {
    requestMessage match {
      case submitRequest:CreateSubmissionRequest =>
// 在这里开始构建驱动程序(也就是包含 SparkContext 的应用程序)的
        // 描述信息,对应 DriverDescription 实例
        // 并向 Master 的 RPC 终端 masterEndpoint 发送请求消息 RequestSubmitDriver
val driverDescription = buildDriverDescription(submitRequest)
        val response = masterEndpoint.askWithRetry[DeployMessages.SubmitDriverResponse](
          DeployMessages.RequestSubmitDriver(driverDescription))
        val submitResponse = new CreateSubmissionResponse
…
}
}
```

3）构建 DriverDescription 的 buildDriverDescription 方法的代码如下。

```
DriverDescription private def buildDriverDescription(request:CreateSubmissionRequest):DriverDescription = {
…
   // 构建 Command 实例,将主类 mainClass 封装到 DriverWrapper(可以通过 jps 查看)
    val command = new Command(
      "org.apache.spark.deploy.worker.DriverWrapper",
```

```
        Seq("{{WORKER_URL}}","{{USER_JAR}}",mainClass) ++ appArgs,// args to the
    DriverWrapper
        environmentVariables,extraClassPath,extraLibraryPath,javaOpts)
    …

  //构建驱动程序的描述信息 DriverDescription
  new DriverDescription(
        appResource,actualDriverMemory,actualDriverCores,actualSuperviseDriver,command)
  }
```

4）Master 接收并处理消息的代码如下。

```
  override def receiveAndReply(context:RpcCallContext):PartialFunction[Any,Unit] = {
…
 // 接收并处理 RequestSubmitDriver 消息
case RequestSubmitDriver(description) => {
      if(state != RecoveryState.ALIVE) {
        val msg = s"${Utils.BACKUP_STANDALONE_MASTER_PREFIX}:$state. " +
          "Can only accept driver submissions in ALIVE state."
        context.reply(SubmitDriverResponse(self,false,None,msg))
      } else {
 //这里的 mainClass 就是被封装的应用程序的主类
 logInfo("Driver submitted " + description.command.mainClass)

          // 创建 Driver 信息,在 Master 中需要调度 Application 和 Driver
          val driver = createDriver(description)
          persistenceEngine.addDriver(driver)
          waitingDrivers += driver
          drivers.add(driver)

          // 开始根据调度机制进行调度
          schedule()
 …
        }
    }
  }
```

5）Master 的 schedule()：调度机制的调度代码如下。

```
  /**
   * Schedule the currently available resources among waiting apps. This method will be called
   * every time a new app joins or resource availability changes
   */
  private def schedule():Unit = {
```

```
        if(state != RecoveryState.ALIVE) { return }
        // Drivers 优先于 executors 调度,逻辑上需要先调度驱动程序,然后再为驱动程序的
        // 具体任务执行分配 Executors
        // Drivers take strict precedence over executors
    // 均衡调度的一种机制
    val shuffledWorkers = Random.shuffle(workers)// Randomization helps balance drivers
        for(worker <- shuffledWorkers if worker.state == WorkerState.ALIVE) {
          for(driver <- waitingDrivers) {
            if(worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {
              // 开始启动 Driver,在该方法中会向 Worker 发送启动 Driver 的请求消息,
              // 详细的消息发送代码请参考源代码
              launchDriver(worker, driver)
              waitingDrivers -= driver
            }
          }
        }
        startExecutorsOnWorkers()
      }
```

6）Worker 上的 Driver 启动的代码如下。

```
    override def receive: PartialFunction[Any, Unit] = synchronized {
  …
      case LaunchDriver(driverId, driverDesc) => {
          logInfo(s"Asked to launch driver $driverId")
  // 构造 DriverRunner 实例
          val driver = new DriverRunner(
            conf,
            driverId,
            workDir,
             sparkHome,
   // 驱动的描述信息:DriverDescription
              driverDesc.copy(command = Worker.maybeUpdateSSLSettings(driverDesc.command,
   conf)),
             self,
             workerUri,
             securityMgr)
          drivers(driverId) = driver

   //启动驱动程序
          driver.start()

```

```
            coresUsed += driverDesc. cores
            memoryUsed += driverDesc. mem
        }
    }
```

（2）提交方式为传统方式

这里的传统方式指的是在 REST 方式之前，Spark 所提供的提交方式。此时会将应用程序的主类等信息封装到 org. apache. spark. deploy. Client 类中，该类和 REST 方式下的封装类 RestSubmissionClient 一样，也会将提交的应用程序封装到 DriverDescription，然后向 Master 发送 RequestSubmitDriver 消息，之后的 Master 消息处理与调度等步骤与 REST 方式一样。详细的执行框架如图 3-5 所示。

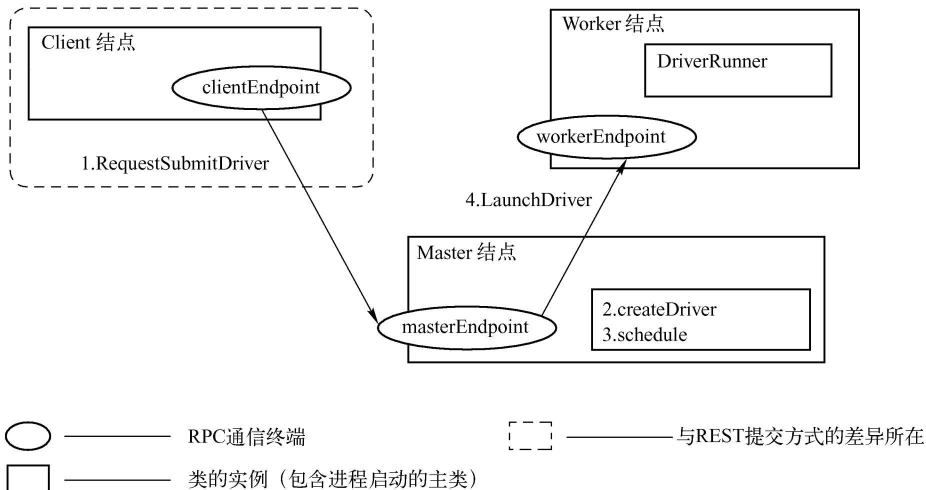


图 3-5　传统提交方式下的详细执行框架

为了体现各个组件间的部署关系，这里以框架图的形式进行描述，相应的，可以从时序图的角度去理解各个类或组件之间的交互关系。其中组件 Master、Worker 和 Client 的标注在方框的左上角，其他方框表示一个具体的实例。其中需要注意的是虚线方框处是该提交方式与 REST 方式之间的差异所在。

和 REST 方式相比，传统的方式只在向 Master 发送提交应用请求的地方不同，其他步骤都一样。其中，Client 是提交应用程序的客户端处，对提交的应用程序进行封装的类。之后各个组件间的交互流程分析如下。

1）第 1 步在 Client 的入口函数 main 中，在该函数中会构建一个与 Master 进行通信的 RpcEndPoint，即 ClientEndpoint 实例。在 ClientEndpoint 实例构建后，会调用该实例的 onStart 方法，然后向 Master 的通信端口发送请求消息 RequestSubmitDriver，然后等待反馈并退出。

2）Master 接收到该消息后，执行第 2 步 createDriver，创建 Driver，创建时需要由 Master 的调度机制（对应第 3 步 schedule），获取分配的资源后，向 Worker（这些 Worker 启动时会注册到 Master 上）的 RPC 终端发送 LaunchDriver 消息。

3）Worker 在 RPC 终端接收到消息后开始处理，实例化一个 DriverRunner，并运行之前

封装的应用程序。

其中2)、3) 和 REST 方式的处理流程一样。

在传统方式提交的执行框架中，主要的源代码及其分析如下。

1) Client 的入口函数 main 的代码如下。



```
/**
  * Executable utility for starting and terminating drivers inside of a standalone cluster.
  */
object Client {
  def main(args:Array[String]){
…
// 构建 RPC 环境实例
      val rpcEnv =
          RpcEnv.create("driverClient", Utils.localHostName(), 0, conf, new SecurityManager
  (conf))

 // 获取 Master 的 RPC 通信终端
      val masterEndpoints = driverArgs.masters.map(RpcAddress.fromSparkURL).
          map(rpcEnv.setupEndpointRef(Master.SYSTEM_NAME, _, Master.ENDPOINT_
  NAME))
 // 通过 RPC 环境实例构建客户端的 RPC 通信终端 ClientEndpoint 实例
  rpcEnv.setupEndpoint("client", new ClientEndpoint(rpcEnv, driverArgs, masterEndpoints,
conf))

 // 等待 Master 反馈并退出
      rpcEnv.awaitTermination()
    }
  }
```

2) ClientEndpoint 的 onStart 方法的代码如下。

```
  override def onStart():Unit = {
    driverArgs.cmd match {
// 这里处理的是启动应用程序的消息
      case "launch" =>
 // 构建 Command 实例,将主类 mainClass 封装到 DriverWrapper(可以通过 jps 查看)
          val mainClass = "org.apache.spark.deploy.worker.DriverWrapper"
    …
          val command = new Command(mainClass,
            Seq("{{WORKER_URL}}", "{{USER_JAR}}", driverArgs.mainClass) ++ driv-
erArgs.driverOptions,
              sys.env, classPathEntries, libraryPathEntries, javaOpts)

```

```
// 将 command 封装到 DriverDescription,可以结合 ApplicationDescription 一起查看
        val driverDescription = new DriverDescription(
          driverArgs.jarUrl,
          driverArgs.memory,
          driverArgs.cores,
          driverArgs.supervise,
          command)

// 发送提交 Driver 的请求(注意,这里的 Driver 可以理解成请求启动
// 驱动程序(Driver Program)的驱动)
        ayncSendToMasterAndForwardReply[SubmitDriverResponse](
          RequestSubmitDriver(driverDescription))

      case "kill" =>
        val driverId = driverArgs.driverId
// 发送停止 Driver 的请求
ayncSendToMasterAndForwardReply[KillDriverResponse](RequestKillDriver(driverId))
      }
    }
```

3.4.3 Master 的部署

Spark 中的各个组件是通过脚本来启动部署的，下面以脚本为切入点开始分析 Master 的部署。

每个组件对应提供了启动的脚本，同时也会提供停止的脚本。停止脚本比较简单，可以自己查看，在此仅分析启动脚本。

1. Master 部署的启动脚本解析

首先来看下 Master 的启动脚本./sbin/start-master.sh，内容如下。

```
# 在脚本的执行结点启动 Master 组件
# Starts the master on the machine this script is executed on.

#如果没有设置环境变量 SPARK_HOME,会根据脚本所在位置自动设置
if [ -z "${SPARK_HOME}" ];then
  export SPARK_HOME= "$(cd "`dirname "$0"`"/..;pwd)"
fi

# NOTE:This exact class name is matched downstream by SparkSubmit.
 # Any changes need to be reflected there.
 # Master 组件对应的类
```



```
CLASS = "org. apache. spark. deploy. master. Master"

#脚本的帮助信息
if [[ "$@" = * --help ]] || [[ "$@" = * -h ]];then
  echo "Usage:. /sbin/start - master. sh [options]"
  pattern = "Usage:"
  pattern += "\|Using Spark's default log4j profile:"
  pattern += "\|Registered signal handlers for"

# 通过脚本 spark - class 执行指定的 Master 类,参数为 --help
   "${SPARK_HOME}"/bin/spark - class$CLASS  --help 2 >&1 | grep  -v "$pattern" 1 >&2
  exit 1
fi

ORIGINAL_ARGS =  "$@"

#控制启动 Master 时,是否同时启动 Tachyon 的 Master 组件
START_TACHYON = false

while(( "$#" ));do
case$1 in
     -- with - tachyon)
      if [ ! - e "${SPARK_HOME}"/tachyon/bin/tachyon ];then
        echo "Error: -- with - tachyon specified,but tachyon not found. "
        exit - 1
      fi
      START_TACHYON = true
      ;;
  esac
shift
done

.    "${SPARK_HOME}/sbin/spark - config. sh"

.    "${SPARK_HOME}/bin/load - spark - env. sh"

#下面的一些参数对应的默认配置属性
if [ "$SPARK_MASTER_PORT" = "" ];then
  SPARK_MASTER_PORT = 7077
fi

// 用于 MasterURL,所以当没有设置时,默认会使用 hostname 而不是 IP 地址
```

```
// 该 MasterURL 在 Worker 注册或应用程序提交时使用
if [ "$SPARK_MASTER_IP" = "" ];then
  SPARK_MASTER_IP = hostname
fi

if [ "$SPARK_MASTER_WEBUI_PORT" = "" ];then
  SPARK_MASTER_WEBUI_PORT = 8080
fi

#通过启动后台进程的脚本 spark - daemon. sh 来启动 Master 组件
 "${SPARK_HOME}/sbin"/spark - daemon. sh start$CLASS 1\
   -- ip$SPARK_MASTER_IP  -- port$SPARK_MASTER_PORT  -- webui - port$SPARK_MASTER_WEBUI_PORT\
  $ORIGINAL_ARGS

#需要时同时启动 Tachyon,此时 Tachyon 是编译在 Spark 内的
if [ "$START_TACHYON" == "true" ];then
   "${SPARK_HOME}"/tachyon/bin/tachyon bootstrap - conf$SPARK_MASTER_IP
   "${SPARK_HOME}"/tachyon/bin/tachyon format  - s
   "${SPARK_HOME}"/tachyon/bin/tachyon - start. sh master
fi
```

通过脚本的简单分析，可以看出 Master 组件是以后台守护进程的方式启动的，对应后台守护进程的启动脚本 spark - daemon. sh，在后台守护进程的启动脚本 spark - daemon. sh 内部，通过脚本 spark - class 来启动一个指定主类的 JVM 进程，相关的代码如下。

```
case "$mode" in
#这里对应的是启动一个 Spark 类
   (class)
      nohup nice  - n "$SPARK_NICENESS" "${SPARK_HOME}"/bin/spark - class $command "$@" >> "$log" 2 >&1 </dev/null &
      newpid = "$!"
      ;;
#这里对应提交一个 Spark 应用程序
   (submit)
      nohup nice  - n "$SPARK_NICENESS" "${SPARK_HOME}"/bin/spark - submit  -- class$command "$@" >> "$log" 2 >&1 </dev/null &
       newpid = "$!"
       ;;

    ( * )
       echo "unknown mode:$mode"
```

```
exit 1
;;
```

通过脚本的分析，可以知道最终执行的是 Master 类（对应的代码为前面的 CLASS = "org. apache. spark. deploy. master. Master"），对应的入口点则是 Master 伴生对象中的 main 方法。下面以该方法作为入口点进一步解析 Master 部署框架。

在部署 Master 组件时，最简单的方式是直接启动脚本，不带任何选项参数，命令如下。

```
./sbin/start-master.sh
```

如需设置选项参数，可以查看帮助信息，根据自己的需要进行设置。

2. Master 的源代码解析

首先查看 Master 伴生对象中的 main 方法，代码如下。

```
def main(argStrings:Array[String]) {
  SignalLogger.register(log)
  val conf = new SparkConf
// 构建参数解析的实例
  val args = new MasterArguments(argStrings,conf)
// 启动 RPC 通信环境及 Master 的 RPC 通信终端
  val (rpcEnv,_,_) = startRpcEnvAndEndpoint(args.host,args.port,args.webUiPort,conf)
  rpcEnv.awaitTermination()
}
```

和其他类，如 SparkSubmit 一样，Master 类的入口点处也包含了对应的参数类 MasterArguments。这里简单介绍一下该类，包括 Spark 属性配置相关的一些解析，对应在其他地方的属性配置部分类似。

下面首先来看一下 MasterArguments 类的主要代码，如下所示。

```
/**
 * Master 的命令行解析器
* Command-line parser for the master.
 */
private[master] class MasterArguments(args:Array[String],conf:SparkConf) {
  // 这里和脚本中的默认配置是一样的
  var host = Utils.localHostName()
  var port = 7077
  var webUiPort = 8080
  var propertiesFile:String = null
```

```
// 这里读取的就是启动脚本中设置的环境变量
// Check for settings in environment variables
  if(System. getenv("SPARK_MASTER_HOST")!= null){
    host = System. getenv("SPARK_MASTER_HOST")
  }
  if(System. getenv("SPARK_MASTER_PORT")!= null){
    port = System. getenv("SPARK_MASTER_PORT"). toInt
  }
  if(System. getenv("SPARK_MASTER_WEBUI_PORT")!= null){
    webUiPort = System. getenv("SPARK_MASTER_WEBUI_PORT"). toInt
  }

// 命令行选项参数的解析
  parse(args. toList)

// 这里会将默认的属性配置设置到 SparkConf 实例中
 // This mutates the SparkConf,so all accesses to it must be made after this line
  propertiesFile = Utils. loadDefaultSparkProperties(conf,propertiesFile)

  if(conf. contains("spark. master. ui. port")){
    webUiPort = conf. get("spark. master. ui. port"). toInt
  }
…
}
```

另外，MasterArguments 中的 printUsageAndExit 方法，对应的就是命令行中的帮助信息。

在解析完 Master 的参数之后，调用 startRpcEnvAndEndpoin 方法启动 RPC 通信环境及 Master 的 RPC 通信终端。该方法的代码如下。

```
  /**
*启动 Master 并返回一个三元组
   * Start the Master and return a three tuple of:
   * (1)The Master RpcEnv
   * (2)The web UI bound port
   * (3)The REST server bound port,if any
   */
 def startRpcEnvAndEndpoint(
     host:String,
      port:Int,
      webUiPort:Int,
      conf:SparkConf):(RpcEnv,Int,Option[Int]) = {
```

```
13.         val securityMgr = new SecurityManager(conf)
14.
15.     // 构建 RPC 通信环境
16.     val rpcEnv = RpcEnv.create(SYSTEM_NAME,host,port,conf,securityMgr)
17.
18.         // 构建 RPC 通信终端,这里会实例化 Master
19.     val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
20.             new Master(rpcEnv,rpcEnv.address,webUiPort,securityMgr,conf))
21.
22.         // 向 Master 的通信终端发送请求,获取绑定的端口号
23.         // 包含 Master 的 web ui 监听端口号和 REST 的监听端口号
24.     val portsResponse = masterEndpoint.askWithRetry[BoundPortsResponse](BoundPortsRequest)
25.        (rpcEnv,portsResponse.webUIPort,portsResponse.restPort)
26.       }
```

由第 19 和第 20 行代码可以看到，代码中创建了一个 Master 实例。首先来看一下 Mater 的定义。

```
1.  private[deploy] class Master(
2.        override val rpcEnv:RpcEnv,
3.        address:RpcAddress,
4.        webUiPort:Int,
5.        val securityMgr:SecurityManager,
6.        val conf:SparkConf)
7.  extends ThreadSafeRpcEndpoint with Logging with LeaderElectable {
```

从第 7 行代码可以看到，Master 继承了 ThreadSafeRpcEndpoint 和 LeaderElectable，其中继承 LeaderElectable 涉及 Master 的 HA（High Availability，高可用性）机制，这里先关注 ThreadSafeRpcEndpoint，继承该类后，Master 作为一个 RpcEndpoint，实例化后首先会调用 onStart 方法，方法的代码如下所示。

```
  override def onStart():Unit = {
    logInfo("Starting Spark master at " + masterUrl)
    logInfo(s"Running Spark version${org.apache.spark.SPARK_VERSION}")

// 构建一个 Master 的 web ui
// 可以查看向 Master 提交的应用程序等信息
    webUi = new MasterWebUI(this,webUiPort)
    webUi.bind()
    masterWebUiUrl = "http://" + masterPublicAddress + ":" + webUi.boundPort

 // 在一个守护线程中启动调度机制,周期性地检查 Worker 是否超时
 // 当 Worker 结点超时后,会修改其状态或从 Master 中移除与其相关的操作
 checkForWorkerTimeOutTask = forwardMessageThread.scheduleAtFixedRate(new Runnable {
```




```
        override def run(): Unit = Utils.tryLogNonFatalError {
          self.send(CheckForWorkerTimeOut)
        }
      }, 0, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)

// 默认情况下会启动 Rest 服务,可以通过该服务向 Master 提交各种请求
    if(restServerEnabled) {
        val port = conf.getInt("spark.master.rest.port", 6066)
        restServer = Some(new StandaloneRestServer(address.host, port, conf, self, masterUrl))
      }
      restServerBoundPort = restServer.map(_.start())

// 度量(Metroics)相关的操作,用于监控
masterMetricsSystem.registerSource(masterSource)
      masterMetricsSystem.start()
      applicationMetricsSystem.start()
       // Attach the master and app metrics servlet handler to the web ui after the metrics systems are
      // started.
      masterMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
      applicationMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)

    //下面是 Master HA 相关的操作
    val serializer = new JavaSerializer(conf)
      val(persistenceEngine_, leaderElectionAgent_) = RECOVERY_MODE match {
…
      }
      persistenceEngine = persistenceEngine_
      leaderElectionAgent = leaderElectionAgent_
    }
```

在启动 Master 组件后，会读取 Master 相关的一些配置信息，这些信息可以直接在 Master 源代码中查看。需要注意的是，由于这些配置信息是在 Master 中使用的，因此需要在启动 Master 组件之前进行配置，保证 Master 实例可以读取正确的配置属性；如果在启动之后修改了这些配置属性，则需要重新启动 Master 组件（对应其他组件的相关配置的修改与此类似）。

Master 组件与 YARN 或 Mesos 类似，其主要功能是资源管理与调度，因此会维护一组提供资源的 Worker 信息和需要调度的应用程序信息，在 Master 组件中会根据一定的调度策略为这些应用程序分配资源注册上来的 Worker 中的资源（当前包含 CPU 与内存）。

下面是维护这些信息的变量定义，为了方便分析，下面会做些简单的顺序调整，代码如下。

```
//维护 Worker 结点信息及相关的映射信息
val workers = new HashSet[WorkerInfo]
  private val idToWorker = new HashMap[String,WorkerInfo]
  private val addressToWorker = new HashMap[RpcAddress,WorkerInfo]

  // 维护全部的 ApplicationInfo,以及等待、完成的 ApplicationInfo 信息
  val apps = new HashSet[ApplicationInfo]
  val waitingApps = new ArrayBuffer[ApplicationInfo]
  private val completedApps = new ArrayBuffer[ApplicationInfo]
    private var nextAppNumber = 0

    // 维护 ApplicationInfo 的各种映射信息
    val idToApp = new HashMap[String,ApplicationInfo]
 private val endpointToApp = new HashMap[RpcEndpointRef,ApplicationInfo]
    private val addressToApp = new HashMap[RpcAddress,ApplicationInfo]
    // Using ConcurrentHashMap so that master - rebuild - ui - thread can add a UI after asyncRe-
 buildUI
    private val appIdToUI = new ConcurrentHashMap[String,SparkUI]

    // 维护全部的 DriverInfo,以及等待、完成的 DriverInfo 信息
    private val drivers = new HashSet[DriverInfo]
    private val completedDrivers = new ArrayBuffer[DriverInfo]
    // Drivers currently spooled for scheduling
    private val waitingDrivers = new ArrayBuffer[DriverInfo]
    private var nextDriverNumber = 0
```

继承 ThreadSafeRpcEndpoint 类之后，作为 RPC 的一个通信终端，Master 会响应各种消息并处理，在此过程中会更新 Master 中维护的信息，另外，响应资源请求时进行资源调度的过程中也会更新这些信息。这里先来分析 Master 对资源的调度过程。调度过程在 schedule 方法中，具体代码如下。

```
/ **
   * 为等待应用调度当前可用的资源
 * 当有新的应用加入或可用资源发生变更时调用该方法
 * Schedule the currently available resources among waiting apps. This method will be called
    * every time a new app joins or resource availability changes.
    * /
  private def schedule( ):Unit = {
     if( state != RecoveryState. ALIVE) {return}
// Drivers 在 Executors 的分配上具有优先权

     // 对当前可分配的 Worker 进行洗牌,可以帮助 Drivers 的均衡部署
```




```
        val shuffledWorkers = Random.shuffle(workers)// Randomization helps balance drivers
        for(worker <- shuffledWorkers if worker.state == WorkerState.ALIVE){
// 对等待队列中的 Drivers 进行调度
for(driver <- waitingDrivers){
    // 只要当前的 Worker 满足 Driver 申请的资源,包括内存大小和可用内核数,
            // 就会在该 Worker 上启动 Driver,并将该 Driver 从等待队列中移除
            // 说明:Driver 是驱动程序,在一个结点上启动,
            // 和 Application 在多个结点上分布式启动不同,因此调度控制比较简单
   if(worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores){
             launchDriver(worker,driver)
             waitingDrivers -= driver
           }
        }
      }
// 开始为应用程序调度资源
startExecutorsOnWorkers()
   }
```

在为等待中的 Driver 分配好资源之后，开始为用户提交的应用程序调度和分配资源，具体的代码在 startExecutorsOnWorkers 方法中，代码如下。

```
  /**
    * 调度并启动 Workers 上的 Executors
 * Schedule and launch executors on workers
    */
  private def startExecutorsOnWorkers():Unit = {
 // 目前仅支持简单的 FIFO 调度器。首先满足等待队列中的第一个应用程序
// 然后满足第二个,依次调度
 // 在 Spark 框架中,调度可以分为多个层次,包含应用程序层面的
 //作业(Job)层面的、作业内的 Stage 层面,以及一个 Stage 中的
  // 任务集(TaskSet)层面的调度
  // 在 Master 组件中的调度,可以认为是面向应用程序
  // (包含 Driver 与 Application)层面的
 // 各个层面上的调度策略可以参考本书的调度章节
  // Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app
     // in the queue,then the second app,etc.
     for(app <- waitingApps if app.coresLeft > 0){
        val coresPerExecutor:Option[Int] = app.desc.coresPerExecutor
        //过滤 Workers,提取出具有足够资源启动一个 Executor 的 Workers
        // 提取之后再根据可用内核数进行倒序,优先使用资源丰富的 Worker 结点
        // 提取的条件包含内存大小和内核数两部分,具体如下。
        // 内存大小的条件
```



```
// Worker 的可用内存大于应用申请的每个 Executor 所需的内存
          //内核数的条件,有以下两种情况。
          // 1)如果应用申请了每个 Executor 所需的内核数
          // 则 Worker 的可用内核数必须大于该内核数
// 2)如果应用对每个 Executor 上的内核数没有要求
// 则 Worker 上只需至少有可用的内核即可
        // Filter out workers that don't have enough resources to launch an executor
          val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
            .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&
              worker.coresFree >= coresPerExecutor.getOrElse(1))
            .sortBy(_.coresFree).reverse
          // 在提取的可用 Workers 上为 Executors 指定分配到的内核数
          val assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spreadOutApps)

    // 在指定了每个 Worker 分配的内核数之后,开始分配并启动 Executor
    // Now that we've decided how many cores to allocate on each worker, let's allocate them
          for(pos <- 0 until usableWorkers.length if assignedCores(pos) > 0) {
            allocateWorkerResourceToExecutors(
              app, assignedCores(pos), coresPerExecutor, usableWorkers(pos))
          }
        }
    }
```

从代码可知，startExecutorsOnWorkers 方法包含下列 3 个步骤。

1）提取满足资源条件的 Worker 队列。

2）指定在每个可用 Worker 上分配的内核数。

3）根据在每个可用 Worker 上分配到的内核数，在各个 Worker 上调度和启动 Executors。

指定在每个可用 Worker 上分配的内核数在 scheduleExecutorsOnWorkers 方法中实现，具体代码如下。

```
  /**
   * 调度在 Workers 上启动的 Executors。返回元素为在每个 Worker 上分配的
   * 内核个数的数组
 * Schedule executors to be launched on the workers
   * Returns an array containing number of cores assigned to each worker
   *
   * 启动 Executors 有两种模式:
   * 第一种是将应用程序的 Executors 部署到尽可能多的 Workers 上
 * 这种模式通常是为了更好地实现数据本地性。默认使用的是第一种模式
    * 第二种和第一种相反。部署到尽可能少的 Worker 上这种模式通常用于计算密集型
      的应用
  * There are two modes of launching executors. The first attempts to spread out an
    * application's executors on as many workers as possible, while the second does the
```




```
         * opposite(i. e. launch them on as few workers as possible). The former is usually better
         * for data locality purposes and is the default.
         *
         * 每个 Executor 上的内核数是可配置的。当显式配置时,如果一个 Worker 上拥有
    * 足够的内存大小和内核数,同一个应用程序就可以在该 Worker 结点上部署
         * 多个 Executors。否则,默认情况下每个 Executor 会使用 Worker 结点上的全部可用
    * 内核,此时这个 Worker 结点上就不能再启动另一个 Executor 了





         * …
         */
      private def scheduleExecutorsOnWorkers(
          app: ApplicationInfo,
          usableWorkers: Array[WorkerInfo],

  // 控制启动 Executors 有两种模式的变量
          spreadOutApps: Boolean): Array[Int] = {
        val coresPerExecutor = app. desc. coresPerExecutor
        val minCoresPerExecutor = coresPerExecutor. getOrElse(1)

      // 如果没有指定每个 Executor 分配的内核数,则一个 Worker 上只启动一个 Executor
        val oneExecutorPerWorker = coresPerExecutor. isEmpty
        val memoryPerExecutor = app. desc. memoryPerExecutorMB
        val numUsable = usableWorkers. length

  // 元素为在每个 Worker 结点上分配到的内核个数的数组
        val assignedCores = new Array[Int](numUsable)// Number of cores to give to each worker
         val assignedExecutors = new Array[Int](numUsable)// Number of new executors on each worker
        // 当前需要分配的内核个数,应用程序需分配的剩余内核个数或
        // 当前 Workers 上全部能分配的内核个数,
        // 取最小值,避免超过当前可分配的总内核数
        var coresToAssign = math. min(app. coresLeft, usableWorkers. map(_. coresFree). sum)

  /** 判断指定索引位置的 Worker 结点是否能为该应用程序启动一个 Executor */

   def canLaunchExecutor(pos: Int): Boolean = {
  // 判断当前需要分配的内核数是否满足每个 Executor 所需的内核数
  val keepScheduling = coresToAssign >= minCoresPerExecutor
```



```
      // 判断当前 Worker 可用内核数减去当前结点已经分配的内核数
// 是否满足每个 Executor 所需的内核数
        val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= minCoresPerExecutor

// 如果允许每个 Worker 结点启动多个 Executors,会相应地启动一个新的 Executors
      // 否则,如果该 Worker 结点上已经启动过一个 Executor,
      // 只需要将更多的内核分配给该 Executor 即可


 val launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0
        if(launchingNewExecutor) {
// 当启动一个新的 Executor 时,除了内核数要满足条件外,
          // 还需要判断新的 Executor 所需的内存是否也满足条件
          // 以及当前应用程序的总 Executor 个数是否满足条件
          val assignedMemory = assignedExecutors(pos) * memoryPerExecutor
          val enoughMemory = usableWorkers(pos).memoryFree - assignedMemory >= memoryPerExecutor
          val underLimit = assignedExecutors.sum + app.executors.size < app.executorLimit
          keepScheduling && enoughCores && enoughMemory && underLimit
        } else {
          // We're adding cores to an existing executor, so no need
          // to check memory and executor limits
          keepScheduling && enoughCores
        }
      }

      // Keep launching executors until no more workers can accommodate any
      // more executors, or if we have reached this application's limits
      // 根据 canLaunchExecutor 条件过滤可用的 Worker 结点
      var freeWorkers = (0 until numUsable).filter(canLaunchExecutor)
      while(freeWorkers.nonEmpty) {
        freeWorkers.foreach { pos =>
          var keepScheduling = true
          while(keepScheduling && canLaunchExecutor(pos)) {
            coresToAssign -= minCoresPerExecutor
            assignedCores(pos) += minCoresPerExecutor

            // If we are launching one executor per worker, then every iteration assigns 1 core
            // to the executor. Otherwise, every iteration assigns cores to a new executor.
            if(oneExecutorPerWorker) {
              assignedExecutors(pos) = 1
```




```
          } else {
            assignedExecutors(pos) += 1
          }

    // spreadOutApps 是指定启动的模式的配置属性,根据该属性判断是否在
          // 当前 Worker 上继续分配内核数,如果 spreadOutApps 为 true 的话,
        // 不再从当前 Worker 结点上继续分配,而是从下一个 Worker 结点上继续分配
  // Spreading out an application means spreading out its executors across as
          // many workers as possible. If we are not spreading out, then we should keep
          // scheduling executors on this worker until we use all of its resources.
          // Otherwise, just move on to the next worker.
          if(spreadOutApps) {
            keepScheduling = false
          }
        }
      }
      freeWorkers = freeWorkers.filter(canLaunchExecutor)
    }
    assignedCores
  }
```

根据在每个可用 Worker 上分配到的内核数，在各个 Worker 调度和启动 Executors，代码如下。

```
  /**
      * 将一个 Worker 结点上的资源分配给一个或多个 Executors
   * Allocate a worker' s resources to one or more executors.
      * @param app the info of the application which the executors belong to
      * @param assignedCores number of cores on this worker for this application
      * @param coresPerExecutor number of cores per executor
      * @param worker the worker info
      */
    private def allocateWorkerResourceToExecutors(
        app: ApplicationInfo,
        assignedCores: Int,
        coresPerExecutor: Option[Int],
        worker: WorkerInfo): Unit = {
      //如果指定了每个 Executor 上的内核个数,就将该 Worker 结点上分配到的内核
      //分给每个 Executor;如果没有指定,则将全部分配到的内核分给一个 Executor


```

```

    //该 Worker 结点上需要启动的 Executor 个数
    val numExecutors = coresPerExecutor. map { assignedCores / _ }. getOrElse(1)
      //每个 Executor 分配到的内核数
        val coresToAssign = coresPerExecutor. getOrElse(assignedCores)
    //为应用程序添加 Executor,并启动 Executor
      for (i <- 1 to numExecutors) {
          val exec = app. addExecutor(worker,coresToAssign)
          launchExecutor(worker,exec)
          app. state = ApplicationState. RUNNING
        }
      }
```

为了更形象地描述 Master 的调度机制，下面通过图 3-6 来介绍抽象的资源调度框架。

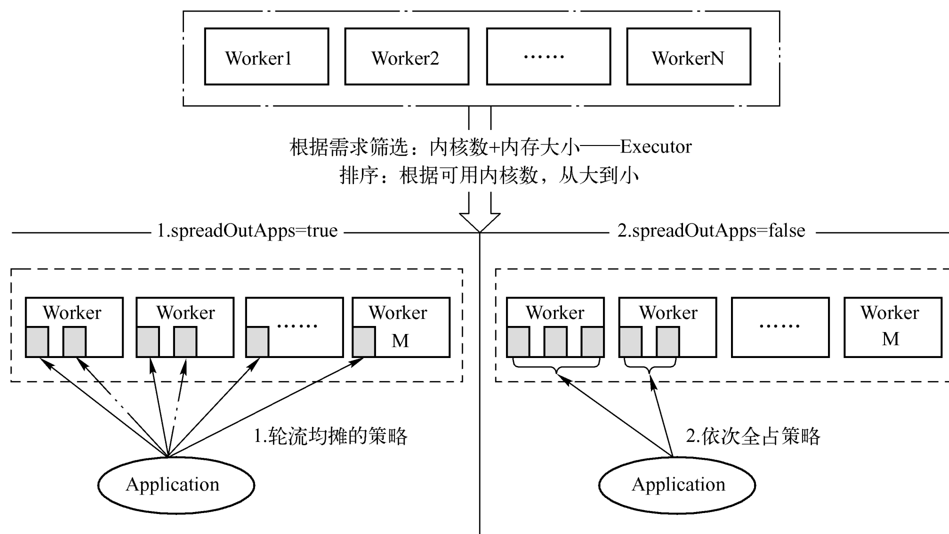


图 3-6　Master 中抽象的资源调度框架

其中，Worker1 到 WorkerN 是集群中全部的 Workers 结点，调度时会根据应用程序请求的资源信息，从全部 Workers 结点中过滤出资源足够的结点，假设可以得到 Worker1 到 WorkerM 的结点。当前过滤的需求是内核数和内存大小足够启动一个 Executor，这是因为 Executor 是集群执行应用程序的单位组件（注意：Executor 和任务（Task）不是同一个概念，对应的任务是在 Executor 中执行的。）

选出可用 Worker 后，会根据内核大小进行排序，这可以理解成是一种基于可用内核排序的、简单的负载均衡策略。然后根据设置的 spreadOutApps 参数，对应指定两种资源分配策略。

1）当 spreadOutApps = true：使用轮流均摊的策略，也就是采用圆桌（Round - Robin）算法，图 3-6 中的虚线表示第一次轮流摊派的资源不足以满足申请的需求，因此开始第二轮摊派，依次轮流均摊直到符合资源需求。

2）当 spreadOutApps = false：使用依次全占策略，依次从可用 Workers 上获取该 Worker 上可用的全部资源，直到符合资源需求。

对应的图 3-6 中 Worker 内部的小方块，在此表示分配的资源的抽象单位，对应资源的条件，理解的关键点在于资源是分配给 Executor 的，因此最终启动 Executor 时，所占用的资源必须满足启动所需的条件。

前面描述了 Workers 上的资源如何分配给应用程序，之后就是正式开始为 Executor 分配资源、并向 Worker 发送启动 Executor 的命令了。根据申请时是否明确指定需要为每个 Executor 分配确定的内核个数，有下列两种情况。

1）明确指定每个 Executor 需要分配的内核个数时：每次分配的是一个 Executor 所需的内核数和内存数，对应在某个 Worker 分配到的总的内核数可能是 Executor 的内核数的倍数，此时，该 Worker 结点上会启动多个 Executor，每个 Executor 需要指定的内核数和内存数（注意该 Worker 结点上分配到的总的内存大小）。

2）未明确指定每个 Executor 需要分配的内核个数时：每次分配 1 个内核，最后所有在某 Worker 结点上分配到的内核都会放到一个 Executor 内（未明确指定内核个数，因此可以一起放入一个 Executor）。因此，最终该应用程序在一个 Worker 上只有一个 Executor（这里指的是针对一个应用程序，当该 Worker 结点上存在多个应用程序时，仍然会为每个应用程序分别启动相应的 Executor。）。

在此强调并补充一下调度机制中使用的两个重要的配置属性。

1）内核个数的控制属性在类 SparkSubmitArguments 的方法 printUsageAndExit 中，如下所示。

```
//指定为所有 Executors 分配的总内核个数
| Spark standalone and Mesos only:
|    --total-executor-cores NUM Total cores for all executors.

//指定需要为每个 Executor 分配的内核个数
|| Spark standalone and YARN only:
|    --executor-cores NUM    Number of cores per executor. (Default:1 in YARN mode,
|                            or all available cores on the worker in standalone mode)
```

2）资源分配策略：数据本地性（数据密集）与计算密集的控制属性，对应的配置属性在 Master 类中，代码如下。

```
private valspreadOutApps = conf.getBoolean("spark.deploy.spreadOut", true)
```

3.4.4 Worker 的部署

Spark 中的各个组件是通过脚本来启动部署的，为了解密 Worker 的部署，以启动 Worker 的脚本为切入点开始分析。

部署 Worker 组件时，最简单的方式是通过配置 Spark 部署目录下的 conf/slaves 文件，然后以批量的方式来启动集群中在该文件中列出的全部结点上的 Worker 实例。启动组件的命令如下。

```
./sbin/start-slaves.sh
```

另一种方式是动态地在某个新增结点上（注意：是新增结点，如果之前已经部署过的话，可以参考后面对启动多个实例的进一步分析）启动一个 Worker 实例，此时可以在该新增的结点上执行如下启动命令。

```
./sbin/start-slave.sh MasterURL
```

其中，参数 MasterURL 表示当前集群中 Master 的监听地址，启动后 Worker 会通过该地址动态注册到 Master 组件，从而实现为集群动态添加 Worker 结点的目的。

1. Worker 部署脚本的解析

部署脚本根据单个结点及多个结点的 Worker 部署，对应有两个脚本：start-slave.sh 和 start-slaves.sh。其中 start-slave.sh 负责在脚本执行结点启动一个 Worker 组件，start-slaves.sh 脚本则会读取配置的 conf/slaves 文件，逐个启动集群中各个 Slave 结点上的 Worker 组件。

（1）首先分析脚本 start-slaves.sh

脚本 start-slaves.sh 提供了批量启动集群中各个 Slave 结点上的 Worker 组件的方法。即，可以在配置好 Slave 结点（即配置好 conf/slaves 文件）之后，通过该脚本一次性全部启动集群中的 Worker 组件。

脚本的代码如下。

```
#根据 conf/slaves 文件指定的每个结点上启动一个 Slave 实例,即 Worker 组件
#Starts a slave instance on each machine specified in the conf/slaves file.

if [ -z "${SPARK_HOME}" ]; then
  export SPARK_HOME="$(cd "`dirname "$0"`"/..; pwd)"
fi

#根据配置信息设置是否需要同时启动 Tachyon 的 Slave 实例
START_TACHYON=false

while (( "$#" )); do
case $1 in
    --with-tachyon)
      if [ ! -e "${SPARK_HOME}/sbin"/../tachyon/bin/tachyon ]; then
        echo "Error: --with-tachyon specified, but tachyon not found."
        exit -1
      fi
      START_TACHYON=true
```




```
        ;;
    esac
shift
done

. "${SPARK_HOME}/sbin/spark-config.sh"
. "${SPARK_HOME}/bin/load-spark-env.sh"

# Find the port number for the master
if [ "$SPARK_MASTER_PORT" = "" ];then
  SPARK_MASTER_PORT=7077
fi

#这里获取 Master 的 IP 信息,需要注意的是,如果当前 SPARK_MASTER_IP 环境变量
#没有配置的话,会通过 hostname 命令来获取,这时候如果不是在 Master 组件所在结
#点启动本脚本的话, Master 的 IP 设置就不一致了,因此为了避免此类错误,建议在
#Master 组件所在的结点上启动该脚本
if [ "$SPARK_MASTER_IP" = "" ];then
  SPARK_MASTER_IP="`hostname`"
fi

# 通过 slaves.sh 脚本启动 Tachyon 的 Slave 实例
if [ "$START_TACHYON" == "true" ];then
  "${SPARK_HOME}/sbin/slaves.sh" cd "${SPARK_HOME}" \;"${SPARK_HOME}/sbin"/../tachyon/bin/tachyon bootstrap-conf "$SPARK_MASTER_IP"

  # set -t so we can call sudo
  SPARK_SSH_OPTS="-o StrictHostKeyChecking=no -t" "${SPARK_HOME}/sbin/slaves.sh" cd "${SPARK_HOME}" \;"${SPARK_HOME}/tachyon/bin/tachyon-start.sh" worker SudoMount \;sleep 1
fi

#通过 slaves.sh 脚本启动 Worker 实例,这里会调用 Worker 启动的另一个
#脚本 start-slave.sh
# Launch the slaves
"${SPARK_HOME}/sbin/slaves.sh" cd "${SPARK_HOME}" \;"${SPARK_HOME}/sbin/start-slave.sh" "spark://$SPARK_MASTER_IP:$SPARK_MASTER_PORT"
```

其中，脚本 slaves.sh 通过 SSH 协议在指定的各个 Slave 结点上执行各种命令，代码比较简单，建议大家自行查看。

在 ssh 启动的 start-slave.sh 命令中，可以看到它的参数是"spark://$SPARK_MASTER_IP:$SPARK_MASTER_PORT"，这实际上就是 Master URL 的值的拼接代码。

（2）继续分析脚本 start－slave. sh

从前面 start－slaves. sh 脚本的分析可以看到，最终是在各个 Slave 结点上执行 start－slave. sh 脚本来部署 Worker 组件的，相应的，就可以通过该脚本动态地为集群添加新的 Worker 组件。



脚本的代码如下。

```

if [ -z "${SPARK_HOME}" ]; then
  export SPARK_HOME="$(cd "`dirname "$0"`"/..; pwd)"
fi

# NOTE: This exact class name is matched downstream by SparkSubmit.
# Any changes need to be reflected there.
# Worker 组件对应的类
CLASS="org.apache.spark.deploy.worker.Worker"

#脚本的用法,其中 master 参数是必选的,Worker 需要与集群的 Master 通信
#这里的 master 对应 Master URL 信息
if [[ $# -lt 1 ]] || [[ "$@" = *--help ]] || [[ "$@" = *-h ]]; then
  echo "Usage: ./sbin/start-slave.sh [options] <master>"
  pattern="Usage:"
  pattern+="\|Using Spark's default log4j profile:"
  pattern+="\|Registered signal handlers for"

  "${SPARK_HOME}"/bin/spark-class $CLASS --help 2>&1 | grep -v "$pattern" 1>&2
  exit 1
fi

. "${SPARK_HOME}/sbin/spark-config.sh"

. "${SPARK_HOME}/bin/load-spark-env.sh"

# First argument should be the master; we need to store it aside because we may
# need to insert arguments between it and the other arguments
MASTER=$1
shift

# Worker 的 web ui 的端口号设置
# Determine desired worker port
if [ "$SPARK_WORKER_WEBUI_PORT" = "" ]; then
  SPARK_WORKER_WEBUI_PORT=8081
fi
```

```
#在结点上启动指定序号的 Worker 实例
# Start up the appropriate number of workers on this machine.
# quick local function to start a worker
function start_instance {
#指定的 Worker 实例的序号,一个结点上可以部署多个 Worker 组件,对应有多个实例
  WORKER_NUM=$1
  shift

  if [ "$SPARK_WORKER_PORT" = "" ]; then
    PORT_FLAG=
    PORT_NUM=
  else
    PORT_FLAG="--port"
    PORT_NUM=$(( $SPARK_WORKER_PORT + $WORKER_NUM - 1 ))
  fi
  WEBUI_PORT=$(( $SPARK_WORKER_WEBUI_PORT + $WORKER_NUM - 1 ))

#和 Master 组件一样,Worker 组件也是使用启动守护进程的 spark-daemon.sh 脚本来
#启动一个 Worker 实例的
  "${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS $WORKER_NUM \
     --webui-port "$WEBUI_PORT" $PORT_FLAG $PORT_NUM $MASTER "$@"
}

#一个结点上部署几个 Worker 组件是由 SPARK_WORKER_INSTANCES 环境变量控制的,
#默认情况下只部署一个实例,start_instance 方法的第一个参数为实例的序号
if [ "$SPARK_WORKER_INSTANCES" = "" ]; then
  start_instance 1 "$@"
else
  for ((i=0;i<$SPARK_WORKER_INSTANCES;i++)); do
    start_instance $(( 1 + $i )) "$@"
  done
fi
```

在手动启动 Worker 实例时，如果需要在一个结点上部署多个 Worker 组件，则需要配置 SPARK_WORKER_INSTANCES 环境变量，否则多次启动脚本部署 Worker 组件时会报错，其原因在于 spark-daemon.sh 脚本的执行控制，这里给出关键代码的简单分析。

首先脚本中有实例是否已经运行的判断，代码如下。

```
run_command() {
  mode="$1"
  shift

```

```
    mkdir -p "$SPARK_PID_DIR"

  #检查记录对应实例的 PID 的文件,如果对应进程已经运行,则会报错
    if [ -f "$pid" ];then
      TARGET_ID="$(cat "$pid")"
      if [[ $(ps -p "$TARGET_ID" -o comm=) =~ "java" ]];then
        echo "$command running as process $TARGET_ID. Stop it first."
        exit 1
      fi
    fi
  }
```

其中，记录对应实例的 PID 的文件相关代码如下。

```
#这是 PID 文件所在的目录,如果没有设置,默认为/tmp
#如果使用了默认目录,可能会出现停止组件失败的信息,
# 原因在于该/tmp 下的文件可能会被系统自动删除
if [ "$SPARK_PID_DIR" = "" ];then
  SPARK_PID_DIR=/tmp
fi

# 这是指定实例编号对应的 pid 文件的路径
#其中 $instance 代表实例编号,因此如果编号相同的话,对应的是同一个文件
pid="$SPARK_PID_DIR/spark-$SPARK_IDENT_STRING-$command-$instance.pid"
```

从上面的分析可以看出，如果不是通过设置 SPARK_WORKER_INSTANCES，然后一次性启动多个 Worker 实例，而是手动一个个地启动的话，对应的，在脚本中每次启动时的实例编号都是 1，在后台守护进程的 spark-daemon.sh 脚本中生成的 pid 就是同一个文件，因此第二次启动时，pid 文件已经存在，此时就会报错（对应停止时也是通过读取 pid 文件获取进程 ID 的，因此自动停止多个实例的话，也需要设置 SPARK_WORKER_INSTANCES）。

2. Worker 的源代码解析

首先查看 Worker 伴生对象中的 main 方法，代码如下。

```
private[deploy] object Worker extends Logging {
  val SYSTEM_NAME = "sparkWorker"
  val ENDPOINT_NAME = "Worker"

  def main(argStrings:Array[String]) {
    SignalLogger.register(log)
    val conf = new SparkConf
  //构建解析参数的实例
    val args = new WorkerArguments(argStrings,conf)
```

```
//启动 RPC 通信环境及 Worker 的 RPC 通信终端
    val rpcEnv = startRpcEnvAndEndpoint(args.host,args.port,args.webUiPort,args.cores,
      args.memory,args.masters,args.workDir,conf = conf)
    rpcEnv.awaitTermination()
  }
}
```

可以看到，Worker 伴生对象中的 main 方法的格式和 Master 基本一致。通过参数的类型 WorkerArguments 来解析命令行参数，具体的代码解析可以参考 Master 结点部署时的 MasterArguments 的代码解析。

另外，MasterArguments 中的 printUsageAndExit 方法对应的就是命令行中的帮助信息。

在解析完 Worker 的参数之后，调用 startRpcEnvAndEndpoint 方法启动 RPC 通信环境及 Worker 的 RPC 通信终端。该方法的代码解析可以参考 Master 结点部署时使用的同名方法的代码解析。

最终会实例化一个 Worker 对象。Worker 也是继承 ThreadSafeRpcEndpoint，对应的也是一个 RPC 的通信终端，实例化该对象后会调用 onStart 方法，该方法的代码如下。

```
override def onStart() {
  //刚启动时 Worker 肯定是未注册的状态
  assert(!registered)
  logInfo("Starting Spark worker %s:%d with %d cores,%s RAM".format(
    host,port,cores,Utils.megabytesToString(memory)))
  logInfo(s"Running Spark version ${org.apache.spark.SPARK_VERSION}")
  logInfo("Spark home:" + sparkHome)

  //构建工作目录
  createWorkDir()

  //启动 Shuffle 服务
  shuffleService.startIfEnabled()

  //和 Master 的 web ui 一样,Worker 也会启动一个 web ui
  webUi = new WorkerWebUI(this,workDir,webUiPort)
  webUi.bind()

  //注册到 Master: Spark 集群是 Master/Slaves 结构,每个 Slave 结点上启动 Worker
  //组件时,都需要向集群中的 Master 进行注册
  registerWithMaster()

  metricsSystem.registerSource(workerSource)
  metricsSystem.start()
  //Attach the worker metrics servlet handler to the web ui after the metrics system is started.
  metricsSystem.getServletHandlers.foreach(webUi.attachHandler)
}
```

其中，createWorkDir( )方法对应构建了该 Worker 结点上的工作目录，后续在该结点上执行的 Application 相关信息都会存放在该目录下。代码如下。

```
private def createWorkDir( ) {
  workDir = Option(workDirPath). map(new File(_)). getOrElse(new File(sparkHome,"work"))
  try {
    //This sporadically fails - not sure why ... !workDir. exists( )&& !workDir. mkdirs( )
    //So attempting to create and then check if directory was created or not.
    workDir. mkdirs( )
  …
  }
}
```

可以看到如果没有设置 workDirPath，默认使用的是 sparkHome 目录下的 work 子目录。对应的 workDirPath 在 Worker 实例化时传入，反推代码可以查到该变量在 WorkerArguments 中设置。相关代码有两处，一处在 WorkerArguments 的主构造体中，代码如下。

```
if (System. getenv("SPARK_WORKER_DIR")! = null) {
  workDir = System. getenv("SPARK_WORKER_DIR")
}
```

即 workDirPath 由环境变量"SPARK_WORKER_DIR"设置。

另外一处在命令行选项解析时设置，代码如下。

```
private def parse(args: List[String]): Unit = args match {
…
  case (" -- work - dir" | " - d")::value::tail =>
    workDir = value
    parse(tail)
…
}
```

即 workDirPath 由启动 Worker 实例时传入的可选项" -- work - dir"设置。

📖 属性配置：通常由命令可选项来动态设置启动时的配置属性，此时配置的优先级高于默认的属性文件及环境变量中设置的属性。

启动 Worker 后的一个关键步骤就是注册到 Master，对应的方法 registerWithMaster( )的代码如下。

```
  private def registerWithMaster( ) {
    //onDisconnected may be triggered multiple times,so don't attempt registration
    //if there are outstanding registration attempts scheduled.
//设置注册重试定时器
registrationRetryTimer match {
      case None =>
```

```
                registered = false
                //尝试注册到所有的 Masters
                registerMasterFutures = tryRegisterAllMasters()
    //控制重试的次数
    connectionAttemptCount = 0
                //构建注册重试定时器,注意初始注册重试定时器的时间设置
    //注册重试定时器会周期性地向 Worker 本身发送 ReregisterWithMaster 消息
                registrationRetryTimer = Some(forwordMessageScheduler.scheduleAtFixedRate(
                  new Runnable {
                    override def run():Unit = Utils.tryLogNonFatalError {
                      Option(self).foreach(_.send(ReregisterWithMaster))
                    }
                  },
                  INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
                  INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
                  TimeUnit.SECONDS))
              case Some(_) =>
                logInfo("Not spawning another attempt to register with the master,since there is an" +
                  " attempt scheduled already.")
          }
      }
```

继续查看 tryRegisterAllMasters()方法，代码如下。

```
    private def tryRegisterAllMasters():Array[JFuture[_]] = {
    //对每个 Master 都提交一次注册请求
          masterRpcAddresses.map { masterAddress =>
            //通过注册线程池提交,注意线程池大小的设置
            registerMasterThreadPool.submit(new Runnable {
              override def run():Unit = {
                try {
                  logInfo("Connecting to master " + masterAddress + "...")
    //获取 Master 的 RPC 通信终端
                  val masterEndpoint =
                     rpcEnv.setupEndpointRef(Master.SYSTEM_NAME,masterAddress,Master.ENDPOINT_
    NAME)
    //向特定的 Master 的 RPC 通信终端发送注册 Worker 的消息
    registerWithMaster(masterEndpoint)
                } catch {
                  case ie:InterruptedException => //Cancelled
                  case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress",e)
                }
              }
            })
          }
      }
```

其中 registerWithMaster（masterEndpoint）向特定 Master 的 RPC 通信终端发送消息，并且在接收到反馈消息后，进一步调用 handleRegisterResponse 方法进行处理。对应的处理代码如下。

```
private def handleRegisterResponse(msg:RegisterWorkerResponse):Unit = synchronized {
  msg match {
    //成功注册该 Worker 结点,设置 registered,修改当前的 Master
    case RegisteredWorker(masterRef,masterWebUiUrl) =>
      logInfo("Successfully registered with master " + masterRef.address.toSparkURL)
      registered = true
      changeMaster(masterRef,masterWebUiUrl)
    //启动周期性心跳发送调度器,在 Worker 生命周期中定期向 Worker 自动发送
    //SendHeartbeat,在 receive 方法中会向 Master 发送心跳
      forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
        override def run():Unit = Utils.tryLogNonFatalError {
          self.send(SendHeartbeat)
        }
      },0,HEARTBEAT_MILLIS,TimeUnit.MILLISECONDS)
//启动工作目录的定期清理调度器,默认情况下该配置属性为 false,
    //需要手动设置,对应的属性名为"spark.worker.cleanup.enabled"
  if (CLEANUP_ENABLED){
        logInfo(
          s"Worker cleanup enabled;old application directories will be deleted in: $ workDir")
        forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
          override def run():Unit = Utils.tryLogNonFatalError {
            self.send(WorkDirCleanup)
          }
          }, CLEANUP_INTERVAL_MILLIS, CLEANUP_INTERVAL_MILLIS,
TimeUnit.MILLISECONDS)
      }

//注册失败则退出
    case RegisterWorkerFailed(message) =>
      if (!registered){
        logError("Worker registration failed:" + message)
        System.exit(1)
      }
//注册的 Master 处于 Standby 状态
    case MasterInStandby =>
      //Ignore.Master not yet ready.
  }
}
```

分析到这一步，已经明确了注册及对注册的反馈信息的处理细节，下面来进一步分析注

册重试定时器的相关处理，注册重试定时器会定期向 Worker 本身发送 ReregisterWithMaster 消息，因此可以在 receive 方法中查看该消息的处理，具体代码如下。

```
override def receive:PartialFunction[Any, Unit] = synchronized {
…
    case ReregisterWithMaster =>
    reregisterWithMaster()
…
}
```

最终会调用 registerWithMaster 方法。

3.4.5 内部交互的消息机制

1. 部署组件之间的交互消息

Spark 是一个分布式集群，各个组件通过事件（消息）响应机制进行协作。所有与集群部署相关的消息都在 DeployMessages 类中定义。读者可以根据发送的消息及其对应反馈的消息进行整理，也可以根据 DeployMessages 类中的定义形式进行归类。

下面根据 DeployMessages 类中的消息定义简单整理成表格的形式，方便读者查阅，如表 3-10 ~ 3-22 所示。

表 3-10 Worker 到 Master 的消息

消　息	描　述
RegisterWorker	Worker 向 Master 发送的注册消息。在启动 Worker 时，首先需要注册到 Master，由 Master 负责管理，并调度 Worker 上的资源
ExecutorStateChanged	Worker 向 Master 发送 Executor 状态变更的消息。Executor 在 Worker 上执行，当状态发送变化时，需要通知 Master
DriverStateChanged	Worker 向 Master 发送 Driver 状态变更的消息。Driver 在 Worker 上执行，当状态发送变化时，需要通知 Master
WorkerSchedulerStateResponse	在收到 MasterChanged 消息后，反馈 Worker 结点上调度状态信息的消息定义
Heartbeat	Worker 向 Master 发送的心跳消息

表 3-11 Master 到 Worker 的消息

消　息	描　述
RegisteredWorker	Master 向 Worker 发送的注册成功的消息
RegisterWorkerFailed	Master 向 Worker 发送的注册失败的消息
MasterInStandby	Master 向 Worker 发送当前处于 Standby 状态的消息
ReconnectWorker	Master 在收到 Worker 结点的心跳消息后反馈的重新连接 Worker 的消息
KillExecutor	Master 向 Worker 发送的停止 Executor 的消息
LaunchExecutor	Master 向 Worker 发送的启动 Executor 的消息
LaunchDriver	Master 向 Worker 发送的启动 Driver 的消息
KillDriver	Master 向 Worker 发送的停止 Driver 的消息
ApplicationFinished	Master 向 Worker 发送的应用程序完成的消息



表 3-12 Worker 内部的消息

消　　息	描　　述
WorkDirCleanup	周期性发送到 Worker 终端，以便清理应用程序的消息
ReregisterWithMaster	Worker 尝试重新注册到 Master 的消息

表 3-13 AppClient 到 Master 的消息

消　　息	描　　述
RegisterApplication	启动驱动程序之后，AppClient 向 Master 发送的注册应用程序的请求
UnregisterApplication	AppClient 在收到 StopAppClient 消息之后，向 Master 发送的取消注册应用程序的请求
MasterChangeAcknowledged	在 Master HA 时，当 Master 开始恢复元数据信息时，会向 AppClient（以及当前注册上来的各个 Workers 结点）发送 Master 变更的消息 MasterChanged，在收到并处理完该消息时，AppClient 向 Master 发送确认的消息 MasterChangeAcknowledged
RequestExecutors	AppClient 向 Master 发送的期望获取的总的 Executor 个数的资源请求
KillExecutors	与 RequestExecutors 类似，向 Master 发送 Executor 的 kill 请求。发送这两个消息的请求的调度可以跟踪到动态资源调度的客户端 ExecutorAllocationClient 中

表 3-14 Master 到 AppClient 的消息

消　　息	描　　述
RegisteredApplication	Master 收到 AppClient 的应用程序的注册消息 RegisterApplication 之后，当成功注册到 Master 时，向 AppClient 反馈的已经注册的消息
ExecutorAdded	当 Master 在调度的 Worker 结点上发送启动 Executor 之后，向 AppClient 发送已经添加 Executor 的消息
ExecutorUpdated	在 Master 中，当收到 Worker 发送来的 Executor 状态变更消息 ExecutorStateChanged，或者在某些场景下移除了注册在 Master 结点上的 Worker 结点时，会向 AppClient 发送这些相关的 ExecutorUpdated 信息（针对前面两种，分别包含 Worker 上特定 Executor 变更或全部 Executor 变更消息）
ApplicationRemoved	当 Master 发现应用程序处理失败或完成时，会处理内部相关的数据信息，同时向 AppClient 发送该应用程序已经移除的消息

表 3-15 DriverClient 与 Master 交互的消息

消　　息	描　　述
RequestSubmitDriver	DriverClient 端向 Master 发送的 Driver 提交请求。另外，在 REST 服务器收到启动应用程序时，也会向 Master 发送该消息
SubmitDriverResponse	Master 在收到 RequestSubmitDriver 消息后反馈的消息
RequestKillDriver	当传入以 Cluster 部署模式提交应用时所封装的 Client 的参数为 kill 而非 launch 时，会通过在 Client 对象的 RPC 通信终端 DriverClient 实例中向 Master 发送 Kill 掉 Driver 的请求。同样在 REST 服务器也会向 Master 发送该消息
KillDriverResponse	Master 在收到 RequestKillDriver 消息后反馈的消息
RequestDriverStatus	当 Client 对象的 RPC 通信终端 DriverClient 实例收到 Master 的反馈信息时（包含启动（launch）与停止（kill）消息），会向 Master 发送获取 Driver 状态信息的请求。同样在 REST 服务器也会向 Master 发送该消息
DriverStatusResponse	Master 在收到 DriverStatusResponse 消息后反馈的消息

表 3-16　AppClient 内部的消息

消　　息	描　　述
StopAppClient	在停止 AppClient 时，内部向自己的 RPC 通信终端发送的停止消息

表 3-17　Maste 到 Worker 和 AppClient 的消息

消　　息	描　　述
MasterChanged	在 Master 启动或 Master HA 中的 Master 发生变更，在恢复持久化的元数据信息时，向对应的 Worker 与 AppClient 发送的 Master 变更消息

表 3-18　MasterWebUI 到 Master 的消息

消　　息	描　　述
RequestMasterState	MasterWebUI 端向 Master 发送的请求查询 Master 状态信息的请求

表 3-19　Master 到 MasterWebUI 的消息

消　　息	描　　述
MasterStateResponse	Master 为 MasterWebUI 发送的状态响应信息，是 RequestMasterState 请求信息的反馈

表 3-20　WorkerWebUI 发送到 Worker 的消息

消　　息	描　　述
RequestWorkerState	WorkerWebUI 端向 Worker 发送的请求查询 Worker 状态信息的请求

表 3-21　Worker to WorkerWebUI 的消息

消　　息	描　　述
WorkerStateResponse	Worker 为 WorkerWebUI 发送的状态响应信息，是 RequestWorkerState 请求信息的反馈

表 3-22　各个地方用于存活检查的消息

消　　息	描　　述
SendHeartbeat	发送心跳消息

2. 驱动程序及其相关的交互消息

除了分布式集群中各个组件之间的消息交互之外，在用户提交应用程序之后，会有一部分与调度相关的交互信息。这部分信息可以在 CoarseGrainedClusterMessage 类的定义文件中找到，读者可以根据发送的消息及其对应反馈的消息进行整理，也可以根据 CoarseGrained-ClusterMessage 类中的定义形式自行整理归类。

3.4.6　Master HA 的部署

1. Master HA 部署脚本的解析

通过前面对 Master 启动脚本的分析，可以知道内部是使用启动守护进程的脚本 spark－daemon. sh 来启动 Master 实例。脚本 spark－daemon. sh 启动时会检查当前实例的 pid 是否已经存在，因此，如果想在单机上模拟出一个 Master HA 的部署集群，需要手动修改 Master 实例的序号，这样才能在单机上同时启动多个 Master 实例。

原有启动 Master 实例的代码如下。

```
" ${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 1 \
  --ip $SPARK_MASTER_IP --port $SPARK_MASTER_PORT --webui-port $SPARK_
  MASTER_WEBUI_PORT \
  $ORIGINAL_ARGS
```

其中，第 1 行传递给脚本 spark-daemon.sh 的实例序号为 1，如果需要单机启动第二个 Master 实例来模拟 Master HA，需要修改该序号 1 为 2，对应的代码修改如下。

```
" ${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 2\
  --ip $SPARK_MASTER_IP --port $SPARK_MASTER_PORT --webui-port $SPARK_
  MASTER_WEBUI_PORT \
  $ORIGINAL_ARGS
```

此时再次使用脚本启动后，对应的 SPARK_MASTER_WEBUI_PORT 端口号由于已经被占用，因此会自动递增，即得到 7078 的端口号，对应新的 Master 的地址和之前的 Master 实例，在端口号上是不一样的。

2. Master HA 源代码解析

这部分内容是在前面 Master 源代码解析的基础之上，进一步详细分析与 Master HA 相关的源代码。

Master 除了继承 ThreadSafeRpcEndpoint，可以作为 RPC 通信终端使用外，还继承了 LeaderElectable。LeaderElectable 是一个特质，代码如下。

```
@DeveloperApi
trait LeaderElectable {
  def electedLeader()
  def revokedLeadership()
}
```

LeaderElectable 特质包含了两个接口，分别是为领导选中处理接口和废除领导层的处理接口。

在 Master 中，与 HA 相关的几个变量代码如下。

```
//初始状态设置为 RecoveryState.STANDBY
private var state = RecoveryState.STANDBY
//Master HA 中用于持久化各种信息的持久化引擎
private var persistenceEngine:PersistenceEngine = _
//Master HA 中用于领导选举的代理
private var leaderElectionAgent:LeaderElectionAgent = _
```

这两个变量的设置位于 onStart 方法中，具体代码如下。

```
…
    val serializer = new JavaSerializer(conf)
    //根据恢复模式设置持久化引擎和领导选举代理
    val (persistenceEngine_,leaderElectionAgent_) = RECOVERY_MODE match {
      case "ZOOKEEPER" =>
        logInfo("Persisting recovery state to ZooKeeper")
//构建一个 ZooKeeper 恢复模式的工厂实例,
        //通过该工厂实例构建出持久化引擎和领导选举代理
val zkFactory =
          new ZooKeeperRecoveryModeFactory(conf,serializer)
        (zkFactory.createPersistenceEngine(),zkFactory.createLeaderElectionAgent(this))
      case "FILESYSTEM" =>
        //构建一个基于文件系统恢复模式的工厂实例,
        //通过该工厂实例构建出持久化引擎和领导选举代理
        val fsFactory =
          new FileSystemRecoveryModeFactory(conf,serializer)
        (fsFactory.createPersistenceEngine(),fsFactory.createLeaderElectionAgent(this))
      case "CUSTOM" =>
        //通过自定义的用于恢复的工厂类名,构造出工厂实例,
        //通过该工厂实例构建出持久化引擎和领导选举代理
        val clazz = Utils.classForName(conf.get("spark.deploy.recoveryMode.factory"))
        val factory = clazz.getConstructor(classOf[SparkConf],classOf[Serializer])
          .newInstance(conf,serializer)
          .asInstanceOf[StandaloneRecoveryModeFactory]
        (factory.createPersistenceEngine(),factory.createLeaderElectionAgent(this))
//默认情况下的恢复模式
case _ =>
        (new BlackHolePersistenceEngine(),new MonarchyLeaderAgent(this))
    }
    persistenceEngine = persistenceEngine_
leaderElectionAgent = leaderElectionAgent_
…
```

从代码中可以看出，当前的 RECOVERY_MODE 有 4 种（包括默认的模式 NONE），分别对应 ZOOKEEPER、FILESYSTEM 和自定义 CUSTOM。每种模式下都会构建出一个对应的工厂实例，由该工厂实例负责构建最终使用的持久化引擎和领导选举代理。

对应配置恢复模式的配置属性的代码如下。

```
private val RECOVERY_MODE = conf.get("spark.deploy.recoveryMode", "NONE")
```

各种模式下都会持久化集群中的各种元数据信息，包含 Application、Driver 和 Worker。下面分别针对这 4 种模式解析其选举机制和持久化过程。

（1）ZOOKEEPER 恢复模式

ZooKeeper 提供了一种领导选举的机制，通过该机制，可以保证集群中只有一个 Master

处于 RecoveryState. Active 状态，其他的 Master 则处于 RecoveryState. Standby 状态。当处于 RecoveryState. Active 状态的 Master 结点出现故障时，ZooKeeper 选举机制会保证选举出新的 Master 结点。

Spark 并不直接使用 ZooKeeper 的 API，而是使用在 ZooKeeper 上进一步封装的 Curator，使用的类如下所示。

```
import org. apache. curator. framework. CuratorFramework
import org. apache. curator. framework. recipes. leader. {LeaderLatchListener, LeaderLatch}
```

CuratorFramework 提供了 high – level 的 API，极大地简化了 ZooKeeper 的使用，并且在 ZooKeeper 上添加了很多特性，包括以下几个。

1）自动连接管理：连接到 ZooKeeper 的 Client 可能会连接中断，而 Curator 会处理这种情况，也就是说，对 Client 来说自动重连是透明的。

2）简洁的 API：简化了原生态的 ZooKeeper 的方法、事件等，提供了更易于使用的接口。

3）Recipe 的实现如下。

- Leader 的选择。
- 共享锁。
- 缓存和监控。
- 分布式的队列。
- 分布式的优先队列。

CuratorFrameworks 通过 CuratorFrameworkFactory 来创建线程安全的 ZooKeeper 的实例。

在启动 ZooKeeper 的选举代理类 ZooKeeperLeaderElectionAgent 时，会执行 start 方法，在该方法中，通过 CuratorFrameworkFactory. newClient 方法来创建 ZooKeeper 的实例，可以传入不同的参数来对实例进行完全控制。获取实例后，必须通过 start( ) 来启动这个实例，在结束时，需要调用 close( )。

对应构建的代码封装在 SparkCuratorUtil 类中，代码如下。

```
  def newClient(
      conf:SparkConf,
      zkUrlConf:String = "spark. deploy. zookeeper. url") :CuratorFramework = {
    val ZK_URL = conf. get(zkUrlConf)
    //通过 CuratorFrameworkFactory. newClient 方法来创建 ZooKeeper 的实例
    val zk = CuratorFrameworkFactory. newClient(ZK_URL,
      ZK_SESSION_TIMEOUT_MILLIS,ZK_CONNECTION_TIMEOUT_MILLIS,
      new ExponentialBackoffRetry(RETRY_WAIT_MILLIS,MAX_RECONNECT_ATTEMPTS))
//构建后调用 start( ) 方法来启动
    zk. start( )
    zk
  }
```

在领导选举代理中会根据当前选举的情况来调用 masterInstance:LeaderElectable 的两个

接口，具体代码如下。

```
private def updateLeadershipStatus(isLeader:Boolean) {
  if (isLeader && status == LeadershipStatus.NOT_LEADER) {
    status = LeadershipStatus.LEADER
    //被选中为领导
    masterInstance.electedLeader()
  } else if (!isLeader && status == LeadershipStatus.LEADER) {
    status = LeadershipStatus.NOT_LEADER
//被废除领导资格
masterInstance.revokedLeadership()
  }
}
```

当被选中为 Leader 时，masterInstance.electedLeader()接口会向 Master 发送 ElectedLeader 消息，然后在 receive 方法中处理该消息，具体代码如下。

```
override def receive:PartialFunction[Any,Unit] = {
    //被选中为领导
    case ElectedLeader => {
    //通过持久化引擎来读取持久化在 ZooKeeper 中的数据,
    //包括 Application、Driver 和 Worker
      val (storedApps,storedDrivers,storedWorkers)= persistenceEngine.readPersistedData(rpcEnv)
//根据恢复的数据设置当前 Master 结点的状态
      state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.isEmpty) {
        RecoveryState.ALIVE
      } else {
        RecoveryState.RECOVERING
      }
      logInfo("I have been elected leader! New state:" + state)
      //当处于恢复过程中时,调用 beginRecovery 方法开始恢复,同时启动定时机制
      //周期性地向自己发送 CompleteRecovery 消息
      if (state == RecoveryState.RECOVERING) {
        beginRecovery(storedApps,storedDrivers,storedWorkers)
        recoveryCompletionTask = forwardMessageThread.schedule(new Runnable {
          override def run():Unit = Utils.tryLogNonFatalError {
            self.send(CompleteRecovery)
          }
        },WORKER_TIMEOUT_MS,TimeUnit.MILLISECONDS)
      }
    }
```

其中，持久化引擎对应为 ZooKeeperPersistenceEngine 类，在该类中实现了将集群的元数据信息持久化/反持久化到 ZooKeeper 的指定路径下。

（2）FILESYSTEM 恢复模式

在这种模式下，选举机制比较简单，Master 启动即认为被选举为领导。对应的持久化引擎只是将持久化/反持久化的位置改成了本地文件系统，也就是将集群中的这些元数据信息保存到本地文件系统，恢复时也从本地文件系统读取。

（3）CUSTOM 恢复模式

这是提供用户自定义恢复 Master 元数据信息的一种模式。通过前面的分析可知，当使用自定义恢复模式时，需要提供一个工厂类，相应的，该工厂类中必须提供构建持久化引擎和领导选举代理的两个接口。由前面的代码可知，自定义的工厂类名在"spark. deploy. recoveryMode. factory"配置属性中设置。

（4）NONE 恢复模式

这是默认的恢复模式，这种模式下不会持久化集群的元数据信息，Master 启动后即认为被选举为领导。

分析完 4 种恢复模式之后，最后给出与 Master HA 相关的一些配置属性，如表 3-23 所示。

表 3-23　与 Master HA 相关的一些配置属性

配置属性	默认值	描　　述
spark. deploy. recoveryMode	NONE	恢复模式（Master 重新启动的模式），有以下 4 种。 1）ZooKeeper 2）FileSystem 3）CUSTOM 4）NONE
spark. deploy. zookeeper. url		ZooKeeper 的 Server 地址
spark. deploy. zookeeper. dir	/spark	ZooKeeper 保存集群元数据信息的文件目录，包括 Application、Driver 和 Worker

Section 3.5 Spark on YARN 的部署模型

3.5.1 部署框架

1. YARN 框架解析

YARN 是 Hadoop 2.0 系统上的资源统一管理平台，其主要作用是实现集群资源的统一管理和调度，它的基本设计思想是将 MRv1 中的 JobTracker 拆分成两个独立的服务：一个全局的资源管理器 ResourceManager（RM）和每个应用程序特有的 ApplicationMaster（AM）。

通常，当集群中存在多种计算框架时，使用 YARN 统一管理资源要比 SparkStandalone 更合适，更利于资源的合理调度与充分利用。

说明：在集群中只需启动 YARN 服务，通常底层也使用 HDFS 存储系统，因此对应也启动 HDFS 服务。此外，Spark Standalone 部署时的 Master 和 Worker 在逻辑上的对应组件已经由 YARN 框架提供，因此不

需要再启动，也就是说，Spark on YARN 部署时，只需要提供可以提交 Spark 应用的客户端即可，之后提交的应用程序由 YARN 框架负责调度、执行及监控等。

和 Spark Standalone 集群一样，在资源管理调度上，YARN 也采用 Master/Slave 结构，在整个资源管理框架中，ResourceManager 对应 Master，NodeManager 对应 Slave。其中，ResourceManager 负责各个 NodeManager 资源的统一管理和调度，NodeManager 则负责启动任务，以及各个任务间的具体资源的隔离。YARN 框架如图 3-7 所示。

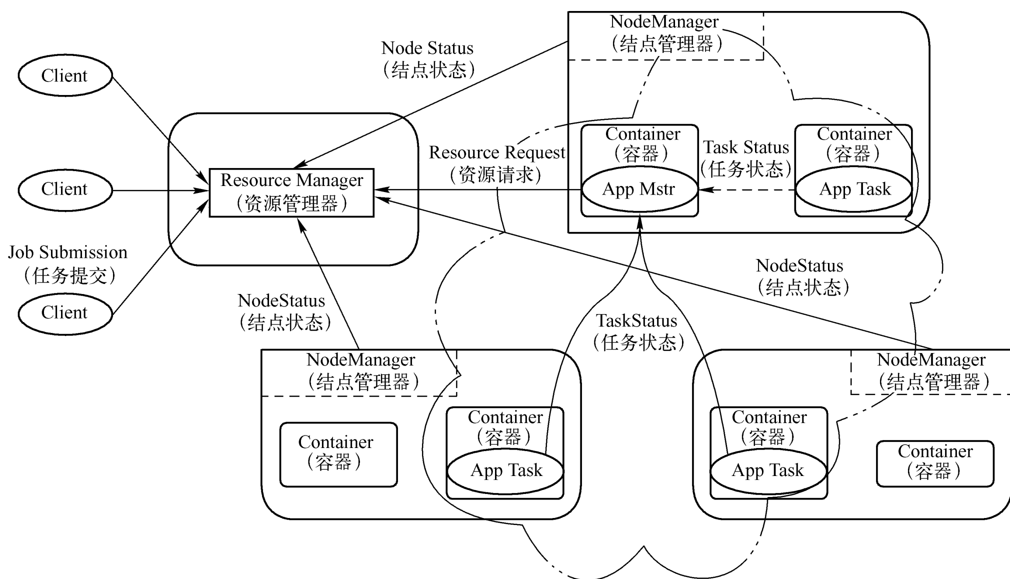


图 3-7　Hadoop YARN 框架的基本结构

下面参考《Hadoop 技术内幕：深入解析 YARN 的架构设计和实现原理》一书，简单描述框架中的主要组件。

（1）ResourceManager（RM）

RM 是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器（Scheduler）和应用程序管理器（Applications Manager，ASM）。

1）调度器。调度器根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运行的应用程序。

需要注意的是，该调度器是一个“纯调度器”，它不再从事任何与具体应用程序相关的工作，比如不负责监控或者跟踪应用的执行状态等，也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务，这些均交由应用程序相关的 ApplicationMaster 完成。调度器仅根据各个应用程序的资源需求进行资源分配，而资源分配单位用一个抽象概念“资源容器”（Resource Container，简称 Container）来表示，Container 是一个动态资源分配单位，它将内存、CPU、磁盘和网络等资源封装在一起，从而限定每个任务使用的资源量。此外，该调度器是一个可插拔的组件，用户可根据自己的需要设计新的调度器，YARN 提供了多种直接可用的调度器，如 Fair Scheduler 和 Capacity Scheduler 等。

2）应用程序管理器。应用程序管理器负责管理整个系统中所有的应用程序，包括应用

程序提交、与调度器协商资源以启动 ApplicationMaster，以及监控 ApplicationMaster 的运行状态并在失败时重新启动它等。

（2）ApplicationMaster（AM）

用户提交的每个应用程序均包含 1 个 AM，主要功能包括以下几个。



- 与 RM 调度器协商以获取资源（用 Container 表示）。
- 将得到的任务进一步分配给内部任务。
- 与 NM 通信以启动/停止任务。
- 监控所有任务的运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

当前 YARN 自带了两个 AM 实现，一个是用于演示 AM 编写方法的实例程序 distributed-shell，它可以申请一定数目的 Container，以并行运行一个 Shell 命令或者 Shell 脚本；另一个是运行 MapReduce 应用程序的 AM-MRAppMaster，此外，一些其他的计算框架对应的 AM 正在开发中，如 Open MPI、Spark 等。

（3）NodeManager（NM）

NM 是每个结点上的资源和任务管理器，一方面，它会定时地向 RM 汇报本结点上的资源使用情况和各个 Container 的运行状态；另一方面，它接收并处理来自 AM 的 Container 启动/停止等各种请求。

（4）Container

Container 是 YARN 中的资源抽象，它封装了某个结点上的多维度资源，如内存、CPU、磁盘和网络等，当 AM 向 RM 申请资源时，RM 为 AM 返回的资源便是用 Container 表示的。YARN 会为每个任务分配一个 Container，且该任务只能使用该 Container 中描述的资源。

需要注意的是，Container 不同于 MRv1 中的 Slot，它是一个动态资源划分单位，是根据应用程序的需求动态生成的。目前，YARN 仅支持 CPU 和内存两种资源，且使用了轻量级资源隔离机制 Cgroups 来进行资源隔离。

2. YARN 上应用程序开发解析

当使用 Spark on YARN 方式提交 Spark 应用程序时，Spark 需要实现 YARN 应用程序接入的接口要求。关于在 YARN 上开发应用程序的相关内容，可以参考《Hadoop 技术内幕：深入解析 YARN 的架构设计和实现原理》一书的相关章节，这里仅给出简单的分析，以便后续对 Spark on YARN 相关部分进行解析。

用户在 YARN 上开发应用时，需要实现以下 3 个模块。

1）Application Client：负责将应用程序提交到 YARN 上，同时，监控应用的运行状态，控制应用的运行。

2）Application Master（AM）：负责整个应用的运行控制，包括向 YARN 注册应用、申请资源和启动容器等，其中，容器是抽象的资源分配单位。

3）Application Worker：负责应用的实际工作，并不是所有的应用都需要编写 Worker。NodeManager 启动 AM 发送过来的容器，容器内部封装了该应用 Worker 运行所需的资源和启动的命令。

要实现上述模块，涉及以下 3 个 RPC 协议。

1）ApplicationClientProtocol：Client – RM 之间的协议，主要用于应用的提交。

2）ApplicationMasterProtocol：AM – RM 之间的协议，AM 通过该协议向 RM 注册并申请

资源。

3）ContainerManagementProtocol：AM－NM 之间的协议，AM 通过该协议控制 NM 启动容器。

3. Spark on YARN 框架解析

Spark on YARN 部署框架的解析，实际上大部分已经由 YARN 框架本身决定了，因此对应的部署框架仅需要关注如何将 Spark 应用程序提交到 YARN 框架上。

通过比较 Spark Standalone 与 YARN 框架上应用程序本身的组成结构，可以看到两者都是采用类似集群 Master/Slave 部署的这种结构。即，无论是 YARN 还是 Spark Standalone 集群，应用程序本身也带有 Master/Slave 形式的组成结构，对应为应用程序的驱动程序和应用程序被拆分的各个逻辑执行单位（可以抽象对应到 Task 或 Executor），其中驱动程序部分对应 Master，而各个逻辑执行单位对应 Slave。将计算与资源拆分开，两者可以对等比较，如图 3-8 所示。

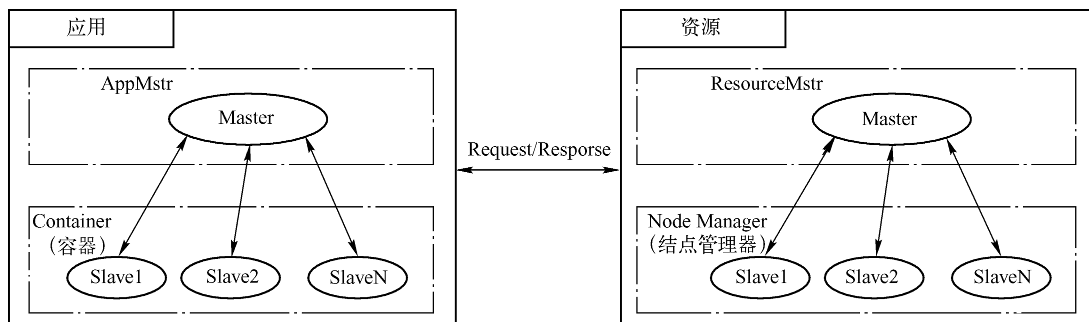


图 3-8　Spark on YARN 框架解析

图 3-8 包含了两种层面上的 Master/Slave 框架，左边对应的是应用层面的，有作为 Master 角色的 AppMstr 和作为 Slave 角色的 Container（具体运行时）。右边对应的是资源层面，同样有作为 Master 角色的 ResourceMstr 和作为 Slave 角色的 Node Manager。如果继续扩展的话，计算、资源和存储等各个层面，甚至于在计算层面上也可以由计算时调度对象的粒度不同而进一步分层，这些层面都可以采用这种中心化的 Master/Slave 框架来实现分布式的计算、资源管理与分配和存储管理等等。中间的 Request/Response 是计算的应用与资源管理间的资源申请与分配，对应可以理解为计算与资源两个解耦之后的组件间的交互。

3.5.2 应用程序的部署

本节主要分析如何将 Spark 应用程序提交到 YARN 上，当 Spark 的应用程序启动之后，应用内部的交互和 Spark Standalone 是一致的。

下面从前面的分析中抽取出 Spark on YARN 模式部署的实例构建信息，如表 3-24 所示。

表 3-24　脚本中的参数与主资源间的对应关系

部署模式（master）	实例对应的类	备　　注
YARN Client	_taskScheduler：YarnScheduler _schedulerBackend：YarnClientSchedulerBackend	YARN 集群管理器 ＋ Client 部署
YARN Cluster	_taskScheduler：YarnClusterScheduler _schedulerBackend：YarnClusterSchedulerBackend	YARN 集群管理器 ＋ Cluster 部署

和 Spark Standalone 一样，Spark on YARN 也存在两种部署模式，包括 Client 部署模式和 Cluster 部署模式。在 Client 的部署模式提交时，直接在提交点运行应用程序，即对应的驱动程序是在当前结点启动的。而在 Spark on YARN 模式部署下，需要通过 ApplicationMaster 管理用户应用程序，因此在该模式下，Client 与 Cluster 两种部署模式下在不同结点启动驱动程序的差异，也就对应了两种部署模式下在集群中启动的 ApplicationMaster 的职责差异。

下面解析这两种部署模式下，Spark 应用程序提交过程的关键源代码。

1. 以 Client 的部署模式提交应用程序

当以 Client 的部署模式提交应用程序时，使用 YarnScheduler 与 YarnClientSchedulerBackend，在 SparkContext 构建出 SparkDeploySchedulerBackend 实例后，然后调用该实例的 start 方法，关键代码如下。

```
/ * *
    * 创建一个 YARN 客户端,用于向 ResourceManager 提交应用程序

    * This waits until the application is running.
    * /
  override def start( ) {
…
//应用程序参数解析
    val args = new ClientArguments( argsArrayBuf. toArray, conf)
    totalExpectedExecutors = args. numExecutors

    //构建 org. apache. spark. deploy. yarn. Client,并通过该类提交应用程序
  client = new Client( args, conf)
    appId = client. submitApplication( )

    //SPARK - 8687:在我们初始化 driver 调度后台程序时候,在 driver 需设置 executor 的
    属性,确保所有必要的属性都已经设置。
    super. start( )

    waitForApplication( )

…
  }
```

从以上代码中可以看出，关键代码位于“org. apache. spark. deploy. yarn. Client”，该类负责应用程序的提交。

2. 以 Cluster 的部署模式提交应用程序

在集群管理器为 YARN、部署模式为 Cluster 时，childMainClass 及对应的 mainClass 的设置如表 3-4 所示。

关于内部的一些环境设置等细节，大家可以查看具体的代码实现。

在 Cluster 部署模式下提交时，封装的主类为"org. apache. spark. deploy. yarn. Client"，同时用户提交的应用程序的主类“args. mainClass”作为参数被封装到该主类中。

当主类为"org. apache. spark. deploy. yarn. Client"时，执行入口点为该类的 main 方法，具体代码如下。

```
    def main(argStrings:Array[String]){
      …
  //Client 的参数解析
      val args = new ClientArguments(argStrings,sparkConf)
  …
  //和 Client 部署模式一样,在入口点实例化一个 Client 实例,并运行 run 方法
      new Client(args,sparkConf).run()
    }
```

结合前面以 Client 的部署模式提交应用程序，可以看到，使用 Client 部署模式提交与使用 Cluster 部署模式提交最终都会实例化一个"org. apache. spark. deploy. yarn. Client"实例。

而在调用该实例的 run 方法时，和 Client 部署模式一样，也会提交应用程序。具体代码如下。

```
    def run():Unit = {
  //通过 Client 实例来提交应用程序
      this. appId = submitApplication()
      …
  }
```

3. Client 与 Cluster 两种部署模式下调用的 submitApplication 方法解析

从 Client 与 Cluster 两种部署模式的入口源代码分析可得，两种模式下最终都调用了 Client 实例的 submitApplication 方法，因此继续分析该方法，具体源代码如下。

```
    /**
     * 向 ResourceManager 提交一个应用程序,来运行 ApplicationMaster

     *
     * The stable Yarn API provides a convenience method (YarnClient#createApplication)for
     * creating applications and setting up the application submission context. This was not
     * available in the alpha API.
     */
    def submitApplication():ApplicationId = {
      var appId:ApplicationId = null
      try {
        …
  //利用 YARN 提供的 API,
```



```
14.            //创建提交客户端 org. apache. hadoop. yarn. client. api. YarnClient
15.            yarnClient. init( yarnConf)
16.            yarnClient. start( )
17.
18.  …
19.
20.            //从 RM 中获取一个新的应用程序,并处理反馈信息
21.            val newApp = yarnClient. createApplication( )
22.  …
23.
24.            //为 AM 创建启动所需的上下文
25.            val containerContext = createContainerLaunchContext( newAppResponse)
26.            val appContext = createApplicationSubmissionContext( newApp,containerContext)
27.
28.            //创建上下文等信息后最终提交并监控应用程序
29.            …
30.            yarnClient. submitApplication( appContext)
31.            appId
32.          } catch {
33.  …
34.          }
35.        }
```

查看第 27 行中 createContainerLaunchContext 方法的关键源代码，如下所示。

```
/* *
      * 创建 ContainerLaunchContext,用于启动 ApplicationMaster 的容器

      * This sets up the launch environment,java options,and the command for launching the AM.
      */
  private def createContainerLaunchContext( newAppResponse:GetNewApplicationResponse)
      :ContainerLaunchContext = {
…
//设置执行的用户主类
  val userClass =
        if (isClusterMode) {
          Seq(" -- class" ,YarnSparkHadoopUtil. escapeForShell( args. userClass) )
        } else {
          Nil
        }
…
  //设置 AM(ApplicationMaster)容器执行的主类
    val amClass =
```

```
19.         if (isClusterMode) {
20.           Utils.classForName("org.apache.spark.deploy.yarn.ApplicationMaster").getName
21.         } else {
22.           Utils.classForName("org.apache.spark.deploy.yarn.ExecutorLauncher").getName
23.         }
24.       …
25.   }
```

从第 10 行与第 18 行代码可知，在 Client 模式下，AM 的类为 ExecutorLauncher，对应用户主类为 Nil，而在 Cluster 模式下，AM 的类为"org. apache. spark. deploy. yarn. ApplicationMaster"，对应用户主类为用户提交的应用程序的主类，即此时将用户提交的主类封装到了"org. apache. spark. deploy. yarn. ApplicationMaster"中。对应在 ExecutorLauncher 的入口方法中会调用"org. apache. spark. deploy. yarn. ApplicationMaster"的入口方法。

"org. apache. spark. deploy. yarn. ApplicationMaster"对应的入口方法中的关键代码如下。

```
final def run():Int = {
…
//根据部署模式运行 Driver 或 Executor 加载器
    if (isClusterMode) {
        runDriver(securityMgr)
      } else {
        runExecutorLauncher(securityMgr)
      }
}
```

在 runDriver 与 runExecutorLauncher 方法中，都会构建 RPC 通信终端，并调用 registerAM，最终通过 YARN API 提供的 org. apache. hadoop. yarn. client. api. AMRMClient 向 RM 注册用户的 ApplicationMaster。

因此在申请提交应用程序并启动之后，"org. apache. spark. deploy. yarn. ApplicationMaster"会在当前提交结点（Client 部署模式）或集群分配的某个结点（Cluster 部署模式）中的容器内启动。

下面继续分析 AM 启动之后，启动实际执行的 Executor 相关代码的源代码解析。

4. Client 与 Cluster 两种部署模式下调用的 Executor 启动的解析

可以从 Executor 在集群中的通信接口出发解析源代码，接口的具体代码如下。

```
/**
   * Executor 所使用的可插拔接口
 * A pluggable interface used by the Executor to send updates to the cluster scheduler.
   */
private[spark] trait ExecutorBackend {
  def statusUpdate(taskId:Long,state:TaskState,data:ByteBuffer)
}
```

Executor 所使用的可插拔接口的具体子类包含 LocalBackend、MesosExecutorBackend 和 CoarseGrainedExecutorBackend，分别用于本地通信、Mesos 集群模式下的通信和其他模式下的通信。

Spark on YANR 集群中使用 CoarseGrainedExecutorBackend 作为 Executor 的通信端口，由 AM 负责向 RM 申请资源并发送到 RM，最终由 RM 负责启动容器，执行 CoarseGrainedExecutorBackend。

从通信接口出发并查看 CoarseGrainedExecutorBackend 通信端最终在 Spark on YARN 集群中创建的位置，可以定位 Executor 的启动位置。

或者通过逐步源代码分析去查看，即通过注册 AM 之后的源代码逐步分析。

在 Client 与 Cluster 两种模式下，都会注册 AM，即调用 registerAM，对应的具体代码如下。

```
    private def registerAM(
        _rpcEnv:RpcEnv,
        driverRef:RpcEndpointRef,
        uiAddress:String,
        securityMgr:SecurityManager) = {
      …
  //通过 YarnRMClient 实例注册,该实例中调用 AMRMClient 实例向 RM 注册,
      //注册成功后返回 YarnAllocator 实例
      allocator = client. register( driverUrl,
        driverRef,
        yarnConf,
        _sparkConf,
        uiAddress,
        historyAddress,
        securityMgr)

      //调用 YarnAllocator 实例的分配资源方法,在该方法中会申请资源并启动
      //Executor 通信接口具体子类
      allocator. allocateResources( )
      reporterThread = launchReporterThread( )
  }
```

Executor 通信接口的具体子类为 CoarseGrainedExecutorBackend，执行的命令封装在 ExecutorRunnable 类。

同样的，Sparkon Mesos 的部署模型与 Spark on YARN 类似，仅仅针对不同的资源管理器 Mesos 提供的应用注册接口，来接入用户的应用程序。Mesos 计算框架是一个集群管理器，提供了有效的、跨分布式应用或框架的资源隔离和共享，可以运行 Hadoop、MPI、Hypertable 和 Spark。使用 ZooKeeper 实现容错复制，使用 Linux Containers 来隔离任务，支持多种资源计划分配。Sparkon Mesos 的部署模型与 Spark on YARN 等其他部署模型相比，主要差异

在于 Mesos 本身的框架，以及对外提供的接入方式等。在此不再赘述。

Section 3.6 小结

本章内容从源代码角度出发，详细解析了 Spark 的不同集群框架，以及在这些框架中的不同部署情况，包括 Local 类的集群部署、Spark Standalone 集群部署、Spark on YARN 集群部署，以及 Spark on Mesos 集群部署下的实现细节。



第 4 章　Spark 调度器（Scheduler）运行机制

Spark 调度器的设计体现得非常简洁清晰和高效，其输入是 Spark RDD，输出是 Spark 执行器（Executor）。正是 Spark 调度器的设计思想极大地区分出了基于 MapReduce 模型的 Hadoop 和基于 DAG 模型的 Spark。本章将分别阐述 Spark 调度器各个重要组件的机制和原理，结合 Spark 调度器的源代码实现，详细解析 Spark 调度器的实现细节，主要内容包括 Spark 运行核心概念、Spark Driver Program 剖析、Spark 作业（Job）的触发、高层的 DAG 调度器（DAGScheduler）、底层的 Task 调度器（TaskScheduler）和调度器的通信终端（Scheduler-Backend）。

Section 4.1 Spark 运行的核心概念

4.1.1 Spark 运行的基本对象

根据前面的阐述可知，RDD 其实就是 Spark 运行的基本对象。关于 RDD 的定义和运行机制，在第 1 章和第 2 章分别给予了详细阐述。这里重新总结下 RDD 的一些重要特征和创建方式，以方便后面对整个 Spark 调度器（Scheduler）的运行机制的理解。

RDD 是一种数据模型，与分布式的共享内存类似。RDD 有以下几个特征。

- 是只读的内存数据，但可以持久化。
- 是可以分区的数据集合。
- 是一个对象，可以调用它的方法执行一些变换操作（Transformation），如 flagMap、filter 等。
- 是可恢复的。与数据多副本的备份方式不同，RDD 的恢复可以通过重复执行变换操作（Transformation）得到。
- 变换操作（Transformation）是延迟操作，只有在真正需要时才执行。

有下列两种方式可以创建 RDD。

- 从持久化的数据上创建，如硬盘和 HDFS 上的文件。
- 从其他 RDD 上通过变换（Transformation）操作创建。

📖 说明：RDD 本身是与编程语言无关的，既可以用 Scala 语言实现，也可以用 Java、Python 等其他编程语言实现。当初决定用 Scala 就是因为用它写的代码简单明了。

4.1.2 Spark 运行框架及各组件的基本运行原理

本章默认讲解的内容都是基于 Spark 的 Standalone 部署模式。在 Standalone 部署模式下，Spark 比在 YARN 和 Mesos 更容易使用，因为不需要其他的东西。如果是基于 Spark 来处理数据，基本上一个 Spark 框架就可以了，无须使用 YARN 或者 Mesos 等（如果掌握了 Standalone 模式，掌握 Yarn 或者 Mesos 是没有问题的，因为它们 80% 的原理都是类似的）。

如图 4-1 所示，SparkContext 在创建 DAGScheduler、TaskScheduler、SchedulerBackend 的同时还会向 Master 注册程序。如果注册没有问题的话，Master 通过 Cluster Manager 会给这个程序分配资源，然后根据 Action 触发 Job。Job 里面有一系列 RDD，DAGScheduler 从后往前推若发现是宽依赖的话，就划分不同的 Stage。Stage 划分完成之后，Stage 提交给底层的调度器 TaskScheduler，TaskScheduler 拿到这个 Task 的集合。因为一个 Stage 内部都是计算逻辑完全一样的任务，只不过是计算的数据不同。TaskScheduler 就会根据数据的本地性，将任务分配到 Executor 上去执行。Executor 在任务运行结束或者出状况时，肯定要向 Driver 汇报。最后运行完毕之后，关闭 SparkContext，同时其创建的那些对象也要关闭掉。

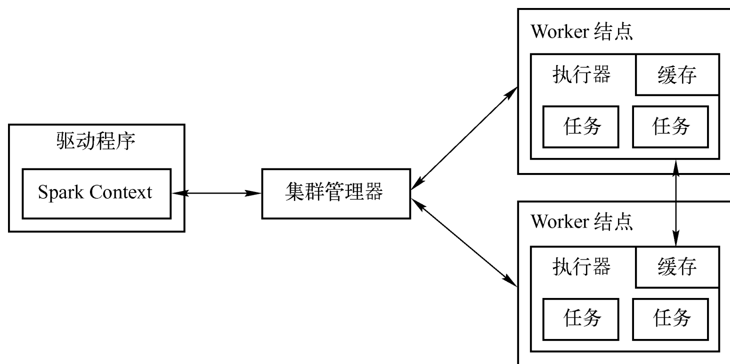


图 4-1　Spark 运行框架图

第 3 章已经介绍了对 Driver、Worker 和 Executor 等 Spark 运行框架的主要组件的基本概念，下面分别对这些组件的基本运行原理给予更为详细的阐述。

Driver 是应用程序运行时的核心，它负责整个作业的调度，同时会向 Master 申请资源完成具体作业的工作过程。所谓应用程序，就是用户编写的 Spark 代码打包后的 jar 包和相关的依赖，其中包括 Driver 功能的代码和分布在集群中多个结点的 Executor 代码。Driver 是驱动 Executor 去工作，Executor 内部是线程池并发地去处理数据分片的。Driver 部分的代码就是 Sparkconf 和 SparkContext 部分。SparkContext 在创建时包含很多内容，包括 DAGScheduler、TaskScheduler、Schedulerbackend 和 Spark - Env 等（一个程序默认有一个 DAGScheduler）。所以一个 Spark Application 通常包含：Driver 端的代码和分布在集群中多个结点上的 Executor

的代码。如 textFile、flatMap 和 map 等可以产生很多 RDD 的方法是具体的业务实现，也就是 Executor 中具体要执行的代码。所有的业务逻辑都是在具体的集群 Worker 上的 Executor 上执行的（前提是代码要发送到集群上）。

Cluster Manager 是集群中获取资源的 Web 服务。在 Spark 的最初阶段并没有 YARN 模式，也没有 Standalone 模式，资源管理服务是 Mesos，后来增加了 YARN，后来为了推广普及产生了 Standalone。最重要的特征是：Spark 的 Application 的运行不依赖于 Cluster Manager。也就是说 Spark 的 Application 注册给 Master，如果注册成功，Master 提前给 Application 分配好了资源，运行过程中根本不需要 Cluster Manager 的参与。Cluster Manager 是可插拔的。这种资源的分配方式是粗粒度的。集群中的具体工作，除了 Cluster Manager、Master 和资源分配器外，这些都是处于主结点上。

Executor 是运行在 Worker 结点上的为当前应用程序开启进程里的处理对象。这个对象负责具体的 Task 运行，是通过线程池并发运行和线程复用的方式。Spark 在一个结点上为当前的程序开启一个 JVM 进程，JVM 进程是线程池的方式，通过线程处理具体的 Task 任务。Executor 是进程里的对象。一个 Worker 默认为当前的应用程序开启一个 Executor（可以配置多个）。Executor 靠线程池中的线程运行 Task 时，肯定会去磁盘或者内存中读写数据。每个 Application 都有自己独立的一批 Executor。

Worker 是集群中任何可以运行 Application 具体的 textFile、flatMap、map、filter 和 reduceByKey 等这些操作代码的结点。Worker 上是不会运行程序代码的，Worker 是管理当前结点 CPU、内存等资源的使用状态，它会接收 Master 分配资源（即 Executor）的指令，会通过 ExecutorRunner 启动一个新进程，进程里面有 Executor。为了便于理解，可以把 Cluster Manager 看成是项目经理，Worker 是工长，项目经理（Cluster Manager）会管理很多工长（Worker），工长下面有很多工人（Executor）。所以，Worker 管理当前结点的计算资源（主要是 CPU 和内存），并接收 Master 的指令，来分配具体的计算资源（在新的进程中分配）。要分配一个新的进程做计算时，ExecutorRunner 相当于一个代理，管理具体新分配的进程，也监控具体的 Executor 所在进程运行的状况。其实就是在 ExecutorRunner 中远程创建出新的进程的。Woker 是一个进程，不会向 Master 汇报当前机器的 CPU 和内存等信息。Worker 发送心跳汇报的信息只有 Workerid。应用程序注册成功时，Master 会给应用程序分配资源，分配时都会记录资源。如果中间 Executor 有丢失的情况，Worker 要向 Master 汇报，然后动态地调整资源。

Section

4.2 Spark Driver Program 剖析

4.2.1 什么是 Spark Driver Program

在 3.4.1 节中，已经介绍过 Spark Driver Program（以下简称 Driver）是运行 Application 的 main 函数，并且新建 SparkContext 实例的程序。其实，初始化 SparkContext 是为了准备 Spark 应用程序的运行环境，在 Spark 中由 SparkContext 负责与集群进行通信、资源的申请，

以及任务的分配和监控等。当 Worker 结点中的 Executor 运行完毕 Task 后，Driver 同时负责将 SparkContext 关闭。通常也可以使用 SparkContext 来代表驱动程序（Driver）。

Driver（或者 SparkContext）整体架构图如图 4-2 所示。

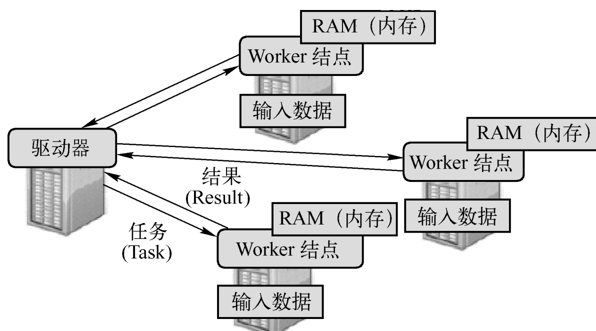


图 4-2　Driver（或者 SparkContext）整体架构图

4.2.2　SparkContext 原理剖析

SparkContext 是用户通往 Spark 集群的唯一入口，可以用来在 Spark 集群中创建 RDD、累加器（Accumulator）和广播变量（Broadcast Variable）。SparkContext 也是整个 Spark 应用程序（Application）中至关重要的一个对象，可以说是整个 Application 运行调度的核心（不是指资源调度）。

SparkContext 的核心作用是初始化 Spark 应用程序运行所需的核心组件，包括高层调度器（DAGScheduler）、底层调度器（TaskScheduler）和调度器的通信终端（SchedulerBackend），同时还会负责 Spark 程序向 Master 注册程序等。

通常，为了测试或者学习用 Spark 开发一个 Application，在 Application 的 main 方法中，最开始几行编写的代码一般总是这样的：首先，会创建 SparkConf 实例，设置 SparkConf 实例的属性以便覆盖 Spark 默认配置文件 spark-env. sh、spark-default. sh 和 log4j. properties 中的参数；然后，SparkConf 实例作为 SparkContext 类的唯一构造参数用来实例化 SparkContext 实例对象。SparkContext 在实例化的过程中会初始化 DAGScheduler、TaskScheduler 和 SchedulerBackend，而当 RDD 的 action 触发了作业（Job）后，SparkContext 会调用 DAGScheduler 将整个 Job 划分成几个小的阶段（Stage），TaskScheduler 会调度每个 Stage 的任务（Task）应该如何处理。另外，SchedulerBackend 管理整个集群中为这个当前的 Application 分配的计算资源（即 Executor）。

如果用汽车来比喻 Spark Application，那么 SparkContext 就是车子的引擎，而 SparkConf 是关于引擎的配置参数。

📖 说明：只可以有一个 SparkContext 实例运行在于一个 JVM 内存中，所以在创建新的 SparkContext 实例之前，必须调用 stop 方法停止当前 JVM 唯一运行的 SparkContext 实例。

4.2.3 SparkContext 源代码解析

下面通过 SparkContext 源代码的解析，来更深入地理解 SparkContext 的原理。

例如，用 Scala 编程语言开发的 “Top N” Application 的开头部分代码如下。

```
//创建 SparkConf 对象
val conf = new SparkConf( )
//设置应用程序的名称,在程序运行的监控界面可以看到名称
conf. setAppName( "Top N!" )
//使程序运行在 Spark 集群
//conf. setMaster( "spark://Master:7077" )
//或者使程序运行在本地,适合于机器配置条件差(如只有 1G 内存)的初学者
conf. setMaster( "local" )
//创建 SparkContext 对象,通过传入 SparkConf 实例来定制 Spark 运行的具体配置参数
val sc = new SparkContext( conf)
…
```

SparkConf 的部分关键代码如下。

```
//SparkConf 负责整个 Spark Application 的配置管理,
//这些配置参数以 key - value 的形式存放在 SparkConf 实例对象的内存中。
//通常情况下,通过 new SparkConf( )来创建一个 SparkConf 实例对象,
//这种方式也会读取所有“spark. * ”形式的 Java 系统属性,
//不过用代码直接设置的配置参数会覆盖系统属性。
//也可以调用 new SparkConf( false)来忽略外部 Java 系统属性设置的参数。
//SparkConf 的 setter 方法支持调用链,例如:
//new SparkConf( ). setMaster( "local" ). setAppName( "My app" )
//注意:SparkConf 的实例对象一旦传给 Spark Application,配置参数就不能再动态改变
class SparkConf( loadDefaults:Boolean) extends Cloneable with Logging {

  import SparkConf. _

  /* * Create a SparkConf that loads defaults from system properties and the classpath */
  def this( ) = this( true)

  private val settings = new ConcurrentHashMap[ String, String] ( )

  if ( loadDefaults) {
    //Load any spark. * system properties
    for ( ( key, value) <- Utils. getSystemProperties if key. startsWith( "spark. " ) ) {
      set( key, value)
    }
```

```
  }
…
  //设置 Application 要连接的 Spark 集群 Master 结点的 URL，例如：
  // – local 表示 Application 运行在本地,有 1 个 Thread 服务
  // – local[4]表示 Application 运行在本地,有 4 个 Thread 服务
  // –“spark://master:7077”表示 Application 运行在 Spark Standalone 集群模式
  def setMaster(master:String):SparkConf = {
    set("spark.master",master)
  }

  //设置应用程序的名称,在程序运行的监控界面可以看到名称
  def setAppName(name:String):SparkConf = {
    set("spark.app.name",name)
  }
…
```

SparkContext 部分初始化关键代码如下。

```
//构造参数 config 是一个描述了 Spark Application 配置的 Spark 对象,它的任何设置都会覆盖默认配置和 System 属性
class SparkContext(config:SparkConf)extends Logging with ExecutorAllocationClient {
…
  //当通过 spark – submit 提交 Spark Application 时,加载系统属性
  def this() = this(new SparkConf())
…
  //返回 SparkContext 配置的副本,这个配置在运行时不能被更改
  def getConf:SparkConf = conf.clone()
…
  //Spark 事件的异步监听 bus
  private[spark] val listenerBus = new LiveListenerBus
…
  //初始化 SparkEnv
  private[spark] def createSparkEnv(
      conf:SparkConf,
      isLocal:Boolean,
      listenerBus:LiveListenerBus):SparkEnv = {
    SparkEnv.createDriverEnv(conf,isLocal,listenerBus,SparkContext.numDriverCores(master))
  }
  private[spark] def env:SparkEnv = _env
…
  //声明调度器的重要组件 SchedulerBackend 实例对象变量
  private[spark] def schedulerBackend:SchedulerBackend = _schedulerBackend
```



```
    private[spark] def schedulerBackend_ = (sb:SchedulerBackend):Unit = {
      _schedulerBackend = sb
    }
    //声明调度器的重要组件 TaskScheduler 实例对象变量
    private[spark] def taskScheduler:TaskScheduler = _taskScheduler
    private[spark] def taskScheduler_ = (ts:TaskScheduler):Unit = {
      _taskScheduler = ts
    }
    //声明调度器的重要组件 DAGScheduler 实例对象变量
    private[spark] def dagScheduler:DAGScheduler = _dagScheduler
    private[spark] def dagScheduler_ = (ds:DAGScheduler):Unit = {
      _dagScheduler = ds
    }
  …
    //根据 Spark 的不同部署模式创建相应的 SchedulerBackend 和 TaskScheduler 实例对象
    val (sched,ts) = SparkContext.createTaskScheduler(this,master)
    _schedulerBackend = sched
    _taskScheduler = ts
    _dagScheduler = new DAGScheduler(this)
  …
    //在 TaskScheduler 设置好 DAGScheduler 对象的引用后,启动 TaskScheduler
    _taskScheduler.start()
  …
    //Spark Application 的唯一标识 ID,例如:
    // - 对于"本地"Spark Application,ID 格式类似'local - 1433865536131'
    // - 对于"YARN"Spark Application,ID 格式类似'application_1433865536131_34483'
    def applicationId:String = _applicationId
    //获取 Spark Application 的标识 ID
    _applicationId = _taskScheduler.applicationId()
    //初始化 BlockManager
    _env.blockManager.initialize(_applicationId)
  …
  }
```

SparkContext 实例化完毕后，就可以调用 SparkContext 的方法来实现各种功能了。下面列出 SparkContext 中几个重要或常用的方法，部分关键代码如下。

```
    //根据给定的 master URL 创建一个 TaskScheduler 具体实现类的实例对象
    //返回 SchedulerBackend 和 TaskScheduler 具体实现类的实例对象
    private def createTaskScheduler(
        sc:SparkContext,
        master:String):(SchedulerBackend,TaskScheduler) = {
      import SparkMasterRegex._
```

```
    //本地运行失败的 Task 不会被重试
    val MAX_LOCAL_TASK_FAILURES = 1

    master match {
      ……
  //从一个本地 Scala 集合得到一个 RDD
  def parallelize[T:ClassTag](
      seq:Seq[T],
      numSlices:Int = defaultParallelism):RDD[T] = withScope {
    assertNotStopped()
    new ParallelCollectionRDD[T](this,seq,numSlices,Map[Int,Seq[String]]())
  }
…
  //读取来自于 HDFS、本地文件系统或者任意一个 Hadoop 支持的文件系统 URI 的文件
  //返回一个由 String 的字符串组成的 RDD
  def textFile(
      path:String,
      minPartitions:Int = defaultMinPartitions):RDD[String] = withScope {
    assertNotStopped()
    hadoopFile(path,classOf[TextInputFormat],classOf[LongWritable],classOf[Text],
      minPartitions).map(pair => pair._2.toString)
  }
…
  //将一个只读广播变量发送给集群中的每个结点
  //该变量只给集群发送一次
  def broadcast[T:ClassTag](value:T):Broadcast[T] = {
    assertNotStopped()
    if (classOf[RDD[_]].isAssignableFrom(classTag[T].runtimeClass)) {
      //用警告而不是异常是为了避免影响用户程序的运行
      //这些程序可能创建了 RDD 广播变量,但是还没有使用它们
      logWarning("Can not directly broadcast RDDs;instead,call collect() and "
        + "broadcast the result (see SPARK-5063)")
    }
    val bc = env.broadcastManager.newBroadcast[T](value,isLocal)
    val callSite = getCallSite
    logInfo("Created broadcast " + bc.id + " from " + callSite.shortForm)
    cleaner.foreach(_.registerBroadcastForCleanup(bc))
    bc
  }
…
  //得到一个特定 RDD 的 partition 的本地性信息
```

```
  private [spark] def getPreferredLocs(rdd:RDD[_],partition:Int):Seq[TaskLocation] = {
    dagScheduler.getPreferredLocs(rdd,partition)
  }
…
  //在给定 RDD 的 partitions 集合上运行函数,并且把结果传给指定的结果处理函数
  //这里是所有 Spark Action 的主入口
  def runJob[T,U:ClassTag](
      rdd:RDD[T],
      func:(TaskContext,Iterator[T]) => U,
      partitions:Seq[Int],
      resultHandler:(Int,U) => Unit):Unit = {
    if (stopped.get()) {
      throw new IllegalStateException("SparkContext has been shutdown")
    }
    val callSite = getCallSite
    val cleanedFunc = clean(func)
    logInfo("Starting job:" + callSite.shortForm)
    if (conf.getBoolean("spark.logLineage",false)) {
      logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
    }
    //将 runJob 转交给 DAGScheduler 去完成
    dagScheduler.runJob(rdd,cleanedFunc,partitions,callSite,resultHandler,localProperties.get)
    progressBar.foreach(_.finishAll())
    rdd.doCheckpoint()
  }
…
  //提交一个要执行的 Job,并且将返回结果放在一个 FutureJob 对象中
  //这里一般由 AsyncRDDAction 类的方法调用
  def submitJob[T,U,R](
      rdd:RDD[T],
      processPartition:Iterator[T] => U,
      partitions:Seq[Int],
      resultHandler:(Int,U) => Unit,
      resultFunc: => R):SimpleFutureAction[R] =
  {
    assertNotStopped()
    val cleanF = clean(processPartition)
    val callSite = getCallSite
    val waiter = dagScheduler.submitJob(
      rdd,
      (context:TaskContext,iter:Iterator[T]) => cleanF(iter),
      partitions,
```



```
91.             callSite,
92.             resultHandler,
93.             localProperties.get)
94.         new SimpleFutureAction(waiter,resultFunc)
95.     }
96.  …
```

Section 4.3 Spark Job 的触发

4.3.1 Job 的逻辑执行（General Logical Plan）

典型的 Job 逻辑执行图如下图 4-3 所示，经过下面四个步骤可以得到最终执行结果：

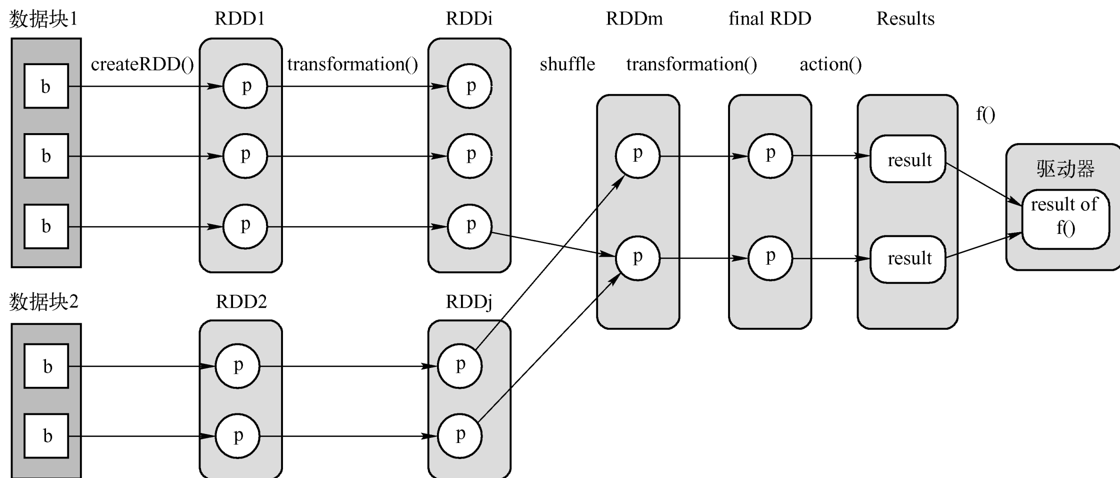


图 4-3　Job 逻辑执行图

1）从数据源（数据源可以是本地 File、内存数据结构、HDFS、HBase 等）读取数据，创建最初的 RDD（createRDD()）；

2）对 RDD 进行一系列的 transformation()操作，每一个 transformation() 会产生一个或多个包含不同类型 T 的 RDD[T]。T 可以是 Scala 里面的基本类型或数据结构，不限于（K,V)。但如果是（K,V)，K 不能是 Array 等复杂类型（因为难以在复杂类型上定义 partition 函数)；

3）对最后的 final RDD 进行 action() 操作，每个 Partition 计算后产生结果 Result。

4）将 Result 回送到 Driver 端，进行最后的 f(list[result]) 计算。例子中的 count() 实际包含了 action() 和 sum() 两步计算。RDD 可以被 Cache 到内存或者 Checkpoint 到磁盘上。RDD 中的 Partition 个数不固定，通常由用户设定。RDD 和 RDD 之间 Partition 的依赖关系可以不是 1 对 1，如图 4-3 所示，既有 1 对 1 关系，也有多对多的关系。

4.3.2 Job 具体的物理执行

Spark Application 里面可以产生 1 个或者多个 Job，例如 spark - shell 默认启动的时候内部就没有 Job，只是作为资源的分配程序，可以在 spark - shell 里面写代码产生若干个 Job，普通程序中一般而言可以有不同的 Action，每一个 Action 一般也会触发一个 Job。

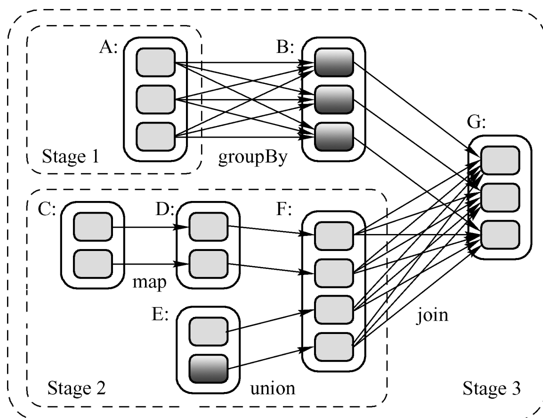
有了 Job 的逻辑执行图，如何生成物理执行图，也就是给定这样一个复杂数据依赖图，如何合理划分 Stage，并确定 Task 的类型和个数？

一个直观的方法是将前后关联的 RDD 组成一个 Stage，每个 Stage 生成一个 Task。这样虽然可以解决问题，但显然效率不高。除了效率问题，这个方法还有一个更严重的问题：大量中间数据需要存储。对于 Task 来说，其执行结果要么要存到磁盘，要么存到内存，或者两者皆有。如果每个箭头都是 Task 的话，每个 RDD 里面的数据都需要存起来，占用空间可想而知。

仔细观察一下逻辑执行图会发现：在每个 RDD 中，每个 Partition 是独立的，也就是说在 RDD 内部，每个 Partition 的数据依赖各自不会相互干扰。因此，一个大胆的想法是将整个流程图看成一个 Stage，为最后一个 finalRDD 中的每个 Partition 分配一个 Task。即 Pipeline 思想：数据用的时候再算，而且数据是流到要计算的位置的。

Spark 算法构造和物理执行时最最基本的核心：最大化 Pipeline！基于 Pipeline 的思想，数据被使用的时候才开始计算，从数据流动的视角来说，是数据流动到计算的位置。实质上从逻辑的角度来看，是算子在数据上流动！从算法构建的角度而言：肯定是算子作用于数据，所以是算子在数据上流动；方便算法的构建！

从 Job 物理执行的角度而言：是数据流动到计算的位置，方便系统最为高效的运行。对于 Pipeline 而言，数据计算的位置就是每个 Stage 中最后的 RDD。也就是说：每个 Stage 中除了最后一个 RDD 算子是真实的以外，前面的算子都是“假”的！由于计算的 Lazy 特性，计算是从后往前回溯，形成 Computing Chain，结果需要首先计算出具体一个 Stage 内部左侧的 RDD 中本次计算依赖的 Partition。如图 4-4 所示：



RDD算子计算：
- 图中三个Stage:Stage1、Stage2、Stage3
- Stage3内部左侧B的计算依赖于G的Partition，如G为3个Partition，那就有3个并行Task，从后往前回溯，B也是3个Partition，3个Task。
- Stage1内部A设置3个Partition，与Stage3的B进行groupBy 操作。
- Stage2内部左侧C、D、F、左侧C的计算依赖于F Partition，F设置2个Partition，2个Task并行计算，从后往前回溯，推出C、D、F也是2个Task，C、D、F形成了map的Pipeline：F设置2个Partition，与E是2个Task并行计算：将C、D与E的计算结果进行Union
- Stage 2 F计算的结果再与Stage 3的B进行Join。

图 4-4　RDD 算子计算过程

总结一下这个过程：整个 Computing Chain 根据数据依赖关系自后向前建立，遇到 ShuffleDependency 后形成 Stage。在每个 Stage 中，每个 RDD 中的 compute( ) 调用 parentRDD. iter( )



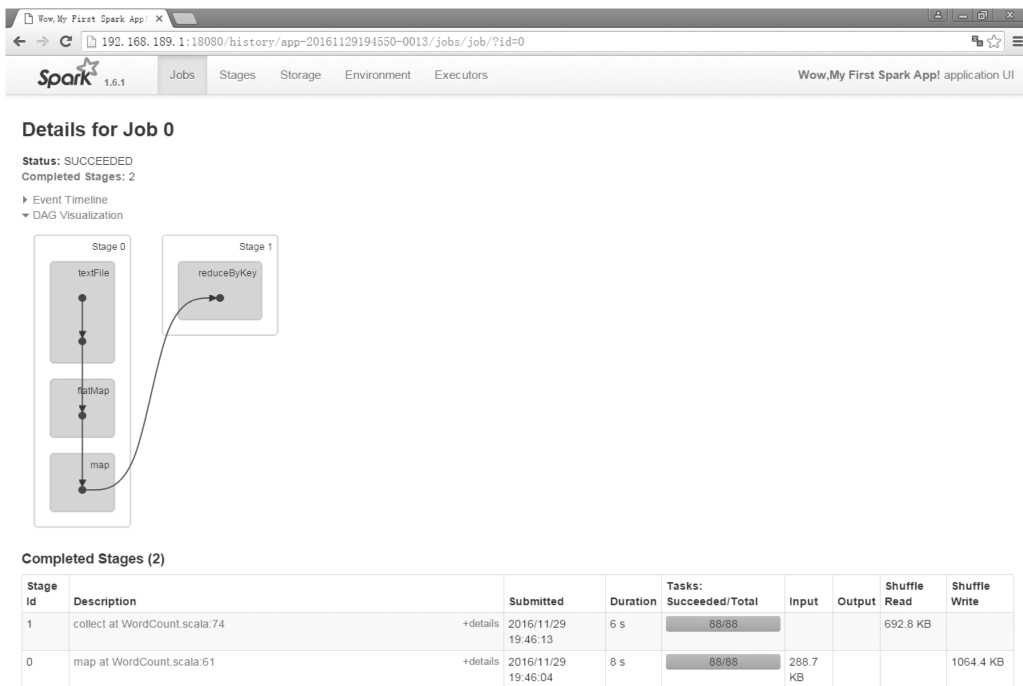
来将 parent RDD 中的 Record 一个个“拿”（fetch）过来。

例如，collect 前面的 RDD 是 transformation 级别的，不会立即执行。从后往前推，回溯时如果是窄依赖则在内存中迭代，否则把中间结果写出到磁盘暂存给后面的计算使用。

依赖分为窄依赖和宽依赖。例如现实生活中，工作依赖一个对象，是窄依赖，依赖很多对象，是宽依赖。窄依赖除了一对一外，还有 range 级别的依赖，依赖固定的个数，随着数据的规模扩大而改变。

如果是宽依赖，DAGScheduler 会划分成不同的 stage，stage 内部是基于内存迭代的，也可以基于磁盘迭代，stage 内部计算的逻辑是完全一样的，只是计算的数据不同而已。具体的任务就是计算一个数据分片，一个 partition 的大小是 128Mb。一个 partition 不是完全精准的等于一个 block 的大小，一般最后一条记录跨两个 block。

如图 4-5 所示，因为 reduceByKey 会产生 shuffle（宽依赖），所以这里有两个 Stage。因为 collect 操作是一个 action，所以会触发 Job。从后往前推，每个 stage 内部都有一系列的任务，下面来看第一个 Stage。Stage 0 内部的 textFile、flatMap、map 默认基于内存迭代，如果内存不够会基于磁盘迭代。



Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at WordCount.scala:74	+details	2016/11/29 19:46:13	6 s	88/88			692.8 KB	
0	map at WordCount.scala:61	+details	2016/11/29 19:46:04	8 s	88/88	288.7 KB			1064.4 KB

图 4-5　Job 0 的所有 Stage 视图

如图 4-6 所示，Stage 0 总共有 88 个任务，图中方框内清楚地告诉用于运行了哪些代码。在具体运行时，stage 内部没有 SparkContext 和 SparkConf。因为它们属于核心 Driver 层面。

如图 4-7 所示，任务都是运行在 Executor 中。默认每台机器为当前的程序分配一个 Executor，这里有 4 个 Worker，所以有 4 个 Executor。可以看到每个 Executor 上的任务运行情况。Worker 本身就是管理 Executor 的。

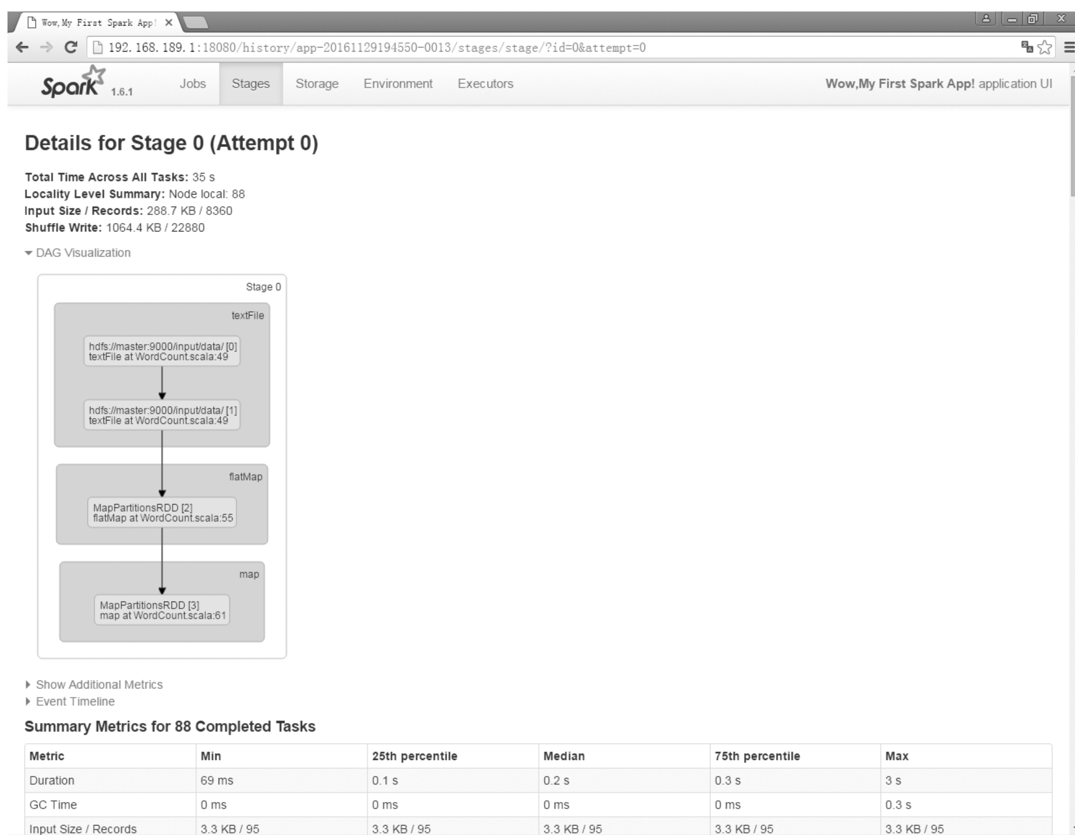


图 4-6　Stage 0 视图

Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
0	worker5:47450	9 s	11	0	11	36.1 KB / 1045	133.1 KB / 2860
1	worker8:51101	8 s	11	0	11	36.1 KB / 1045	133.1 KB / 2860
2	worker6:32976	8 s	7	0	7	23.0 KB / 665	84.7 KB / 1820
3	worker4:36721	8 s	12	0	12	39.4 KB / 1140	145.1 KB / 3120

图 4-7　Worker 中的 Executor 级别的任务运行视图

如图 4-8 所示，Tasks 运行在不同的机器上。

Spark 程序的运行有两种部署方式：Client 和 Cluster。

默认情况下建议使用 Client 模式，此模式下可以看到更多的交互性信息及运行过程的信息。此时要专门使用一台机器来提交 Spark 程序，配置和普通的 Worker 配置一样，而且要和 Cluster Manager 在同样的网络环境中，因为要指挥所有的 Worker 去工作，Worker 里的线程要和 Driver 不断地交互。由于 Driver 要驱动整个集群，频繁地与所有为当前程序分配的 Executor 去交互，频繁地进行网络通信，所以必须在同样的网络中。

也可以指定部署方式为 Cluster，这样 Driver 会由 Master 决定在 Worker 中的某一台机器。Master 为用户分配的第一个 Executor 就是 Driver 级别的 Executor。不推荐学习和开发时使用 Cluster，因为 Cluster 无法直接看到一些日志信息，所以建议使用 Client 方式。

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	2	0	SUCCESS	NODE_LOCAL	5 / worker2	2016/11/29 19:46:05	3 s	0.2 s	3.3 KB (hadoop) / 95	13 ms	12.1 KB / 260	
1	1	0	SUCCESS	NODE_LOCAL	1 / worker8	2016/11/29 19:46:05	2 s	0.1 s	3.3 KB (hadoop) / 95	24 ms	12.1 KB / 260	
2	3	0	SUCCESS	NODE_LOCAL	7 / worker1	2016/11/29 19:46:05	3 s	0.1 s	3.3 KB (hadoop) / 95	13 ms	12.1 KB / 260	
3	5	0	SUCCESS	NODE_LOCAL	2 / worker6	2016/11/29 19:46:05	3 s	0.3 s	3.3 KB (hadoop) / 95	38 ms	12.1 KB / 260	
4	6	0	SUCCESS	NODE_LOCAL	4 / worker3	2016/11/29 19:46:05	2 s	0.1 s	3.3 KB (hadoop) / 95	6 ms	12.1 KB / 260	
5	0	0	SUCCESS	NODE_LOCAL	0 / worker5	2016/11/29 19:46:04	2 s	0.1 s	3.3 KB (hadoop) / 95	11 ms	12.1 KB / 260	
6	7	0	SUCCESS	NODE_LOCAL	3 / worker4	2016/11/29 19:46:05	2 s	0.1 s	3.3 KB (hadoop) / 95	13 ms	12.1 KB / 260	
7	8	0	SUCCESS	NODE_LOCAL	3 / worker4	2016/11/29 19:46:10	0.3 s		3.3 KB (hadoop) / 95	21 ms	12.1 KB / 260	
8	10	0	SUCCESS	NODE_LOCAL	4 / worker3	2016/11/29 19:46:10	0.3 s		3.3 KB (hadoop) / 95	6 ms	12.1 KB / 260	
9	4	0	SUCCESS	NODE_LOCAL	6 / worker7	2016/11/29 19:46:05	2 s	0.1 s	3.3 KB (hadoop) / 95	11 ms	12.1 KB / 260	
10	9	0	SUCCESS	NODE_LOCAL	0 / worker5	2016/11/29 19:46:10	0.3 s		3.3 KB (hadoop) / 95	3 ms	12.1 KB / 260	
11	11	0	SUCCESS	NODE_LOCAL	1 / worker8	2016/11/29 19:46:10	0.2 s		3.3 KB (hadoop) / 95	5 ms	12.1 KB / 260	
12	14	0	SUCCESS	NODE_LOCAL	7 / worker1	2016/11/29 19:46:11	0.2 s		3.3 KB (hadoop) / 95	7 ms	12.1 KB / 260	
13	15	0	SUCCESS	NODE_LOCAL	0 / worker5	2016/11/29 19:46:11	0.3 s		3.3 KB (hadoop) / 95	12 ms	12.1 KB / 260	
14	13	0	SUCCESS	NODE_LOCAL	3 / worker4	2016/11/29 19:46:11	0.2 s		3.3 KB (hadoop) / 95	13 ms	12.1 KB / 260	
15	16	0	SUCCESS	NODE_LOCAL	4 / worker3	2016/11/29	0.2 s		3.3 KB (hadoop) / 95	2 ms	12.1 KB / 260	

图 4-8　Task 级别的运行视图

4.3.3　Job 触发流程源代码解析

对于 Spark Job 触发流程的源代码，下面以 RDD 的 count( ) 为例来讲解。RDD 的 count 方法代码如下。

```
/ * *
  * 返回 RDD 中元素的数量
  * /
def count( ):Long = sc. runJob( this, Utils. getIteratorSize _). sum
```

从上面的代码可以看出，count 方法触发 SparkContext 的 runJob 方法的调用。SparkContext 的 runJob 方法代码如下。

```
/ * *
  * 触发一个 Job 处理一个 RDD 的所有 partitions,并且把处理结果返回到一个数组
  * /
def runJob[T,U:ClassTag](rdd:RDD[T],func:Iterator[T] => U):Array[U] = {
   runJob(rdd,func,0 until rdd. partitions. length)
}
```

进入 SparkContext 的 runJob 方法的同名重载方法，该方法的代码如下。

```
/ * *
  * 触发一个 Job 处理一个 RDD 的指定部分的 partitions,并且把处理结果返回到一个数组。
  * 比第一个 runJob 方法多了一个 partitions 数组参数
  * /
```



```
def runJob[T,U:ClassTag](
      rdd:RDD[T],
      func:Iterator[T] => U,
      partitions:Seq[Int]):Array[U] = {
   val cleanedFunc = clean(func)
   runJob(rdd,(ctx:TaskContext,it:Iterator[T]) => cleanedFunc(it),partitions)
}
```

再进入 SparkContext 的 runJob 方法的另一个同名重载方法，该方法的代码如下。

```
/**
  * 触发一个 Job 处理一个 RDD 的指定部分的 partitions,并且把处理结果返回到一个数组
  * 比第一个 runJob 方法多了一个 partitions 数组参数,并且 func 的类型不同
  */
def runJob[T,U:ClassTag](
      rdd:RDD[T],
      func:(TaskContext,Iterator[T]) => U,
      partitions:Seq[Int]):Array[U] = {
   val results = new Array[U](partitions.size)
   runJob[T,U](rdd,func,partitions,(index,res) => results(index) = res)
   results
}
```

最后一次进入 SparkContext 的 runJob 方法的另一个同名重载方法，该方法的代码如下。

```
/**
  * 触发一个 Job 处理一个 RDD 的指定部分的 partitions,并把处理结果给指定的 handler 函数
  * 这是 Spark 所有 Action 的主入口
  */
def runJob[T,U:ClassTag](
      rdd:RDD[T],
      func:(TaskContext,Iterator[T]) => U,
      partitions:Seq[Int],
      resultHandler:(Int,U) => Unit):Unit = {
   if (stopped.get()) {
      throw new IllegalStateException("SparkContext has been shutdown")
   }
   //记录了方法调用的方法栈
   val callSite = getCallSite
   //清除闭包,为了函数能够序列化
   val cleanedFunc = clean(func)
   logInfo("Starting Job:" + callSite.shortForm)
   if (conf.getBoolean("spark.logLineage",false)) {
      logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
   }
   //向高层调度器(DAGScheduler)提交 Job,从而获得 Job 执行结果
```

```
22.     dagScheduler.runJob(rdd,cleanedFunc,partitions,callSite,resultHandler,localProperties.get)
23.     progressBar.foreach(_.finishAll())
24.     rdd.doCheckpoint()
25.   }
```

关于高层调度器（DAGScheduler）的原理和源代码解析将在下面的章节中详细讲解。

Section 4.4 高层的 DAG 调度器（DAGScheduler）

4.4.1 DAG 的定义

在 2.2 节中介绍了 RDD DAG 构建了 RDD 的数据流，即 RDD 的各个分区的数据是从哪来的；还进一步分析了 RDD DAG 的生成机制和 RDD DAG 的逻辑视图。

RDD DAG 还构建了基于数据流之上的操作算子流，即 RDD 的各个分区的数据总共会经过哪些 Transformation 和 Action 这两种类型的一系列操作的调度运行，从而 RDD 先被 Transformation 操作转换为新的 RDD，然后被 Action 操作将结果反馈到 Driver Program 或存储到外部存储系统上。

上面提到的一系列操作的调度运行其实是 DAG 提交给 DAGScheduler 来解析完成的。DAGScheduler 是面向 Stage 的高层级的调度器，DAGScheduler 把 DAG 拆分成很多 Tasks，每组 Tasks 都是一个 Stage，解析时是以 Shuffle 为边界反向解析构建 Stage（参考 4.4.3 节和 4.4.4 节），每当遇到 Shuffle 就会产生新的 Stage，然后以一个个 TaskSet（每个 Stage 封装一个 TaskSet）的形式提交给底层调度器 TaskScheduler（参考 4.5 节）。另外，DAGScheduler 需要记录哪些 RDD 被存入磁盘等物化动作，同时要寻求 Task 的最优化调度，如在 Stage 内部数据的本地性等（参考 4.4.5 节）。DAGScheduler 还需要监视因为 Shuffle 跨结点输出可能导致的失败，如果发现这个 Stage 失败，可能就要重新提交该 Stage（参考 5.3.3 节的相关内容）。

由此可见，为了更好地理解 Spark 高层调度器 DAGScheduler，除了已经提到过的 RDD（参考 1.2.1 节的相关内容）、Application（参考 3.2 节的相关内容）、Driver Program（参考 4.2.1 节的相关内容）和 Job（参考 4.3 节的相关内容）这些概念以外，还需要了解以下几个概念。

1）Stage：一个 Job 需要拆分成多组任务来完成，每组任务由 Stage 封装。跟一个 Job 的所有涉及的 PartitionRDD 类似，Stage 之间也有依赖关系。

2）TaskSet：一组任务就是一个 TaskSet，对应一个 Stage。其中，一个 TaskSet 的所有 Task 之间没有 Shuffle 依赖，因此互相之间可以并行运行。

3）Task：一个独立的工作单元，由 Driver Program 发送到 Executor（参阅第 5 章）上去执行。通常情况下，一个 Task 处理 RDD 的一个 Partition 的数据。根据 Task 返回类型的不同，Task 又分为 ShuffleMapTask 和 ResultTask。

4.4.2 DAG 的实例化

在 Spark 源代码中，DAGScheduler 是在整个 Spark Application 的入口即 SparkContext 中声

明并实例化的。在实例化 DAGScheduler 之前，已经实例化了 SchedulerBackend 和底层调度器 TaskScheduler，而 SchedulerBackend 和 TaskScheduler 是通过 SparkContaxt 的方法 createTaskScheduler 实例化的。DAGScheduler 在提交 TaskSet 给底层调度器时是面向 TaskScheduler 接口的，这符合面向对象中依赖抽象而不依赖具体实现的原则，带来底层资源调度器的可插拔性，使得 Spark 可以运行在众多的资源部署模式上，例如 Standalone、YARN、Mesos、Local，以及其他自定义的布署模式。

SparkContext 源代码中相关的代码如下。

```
…
@ volatile private var _dagScheduler: DAGScheduler = _

…
private[spark] def dagScheduler: DAGScheduler = _dagScheduler
private[spark] def dagScheduler_ = (ds: DAGScheduler): Unit = {
  _dagScheduler = ds
}

…
val (sched, ts) = SparkContext. createTaskScheduler(this, master)
_schedulerBackend = sched
_taskScheduler = ts

//实例化 DAGScheduler 时传入当前的 SparkContext 实例化对象
_dagScheduler = new DAGScheduler(this)

//启动 TaskScheduler 实例对象,等待 DAGScheduler 提交 TaskSet 给 TaskScheduler
_taskScheduler. start()
…
```

DAGScheduler 源代码中相关的代码如下。

```
  private[spark]
  class DAGScheduler(
      private[scheduler] val sc: SparkContext,
      private[scheduler] val taskScheduler: TaskScheduler,
      listenerBus: LiveListenerBus,
      mapOutputTracker: MapOutputTrackerMaster,
      blockManagerMaster: BlockManagerMaster,
      env: SparkEnv,
      clock: Clock = new SystemClock())
    extends Logging {

    def this(sc: SparkContext, taskScheduler: TaskScheduler) = {
      this(
        sc,
```

```
            taskScheduler,
            sc.listenerBus,
            sc.env.mapOutputTracker.asInstanceOf[MapOutputTrackerMaster],
            sc.env.blockManager.master,
            sc.env)
    }

    //SparkContext 实例化 DAGScheduler 的入口,利用传入的 SparkContext 实例对象来设
    //置将要提交 TaskSet 的 TaskScheduler 实例对象的引用
def this(sc:SparkContext) = this(sc,sc.taskScheduler)

    //TaskScheduler 实例对象也要设置提交给它的 TaskSet 的 DAGScheduler 实例对象的引用
    taskScheduler.setDAGScheduler(this)
    …
}
```

4.4.3 DAGScheduer 划分 Stage 的原理

Spark 在分布式环境下将数据分区，然后将作业转化为 DAG，并分阶段进行 DAG 的调度和任务的分布式并行处理。DAG 将调度提交给 DAGScheduler，DAGScheduler 调度时会根据是否需要经过 Shuffle（具体的 Shuffle 运行机制参考第 7 章）过程将 Job 划分为多个 Stage。

为了方便理解 DAGScheduler 划分 Stage 的原理，下面来回顾一下 2.2 节中的一个典型的 DAG 划分 Stage 示意图，如图 4-9 所示。

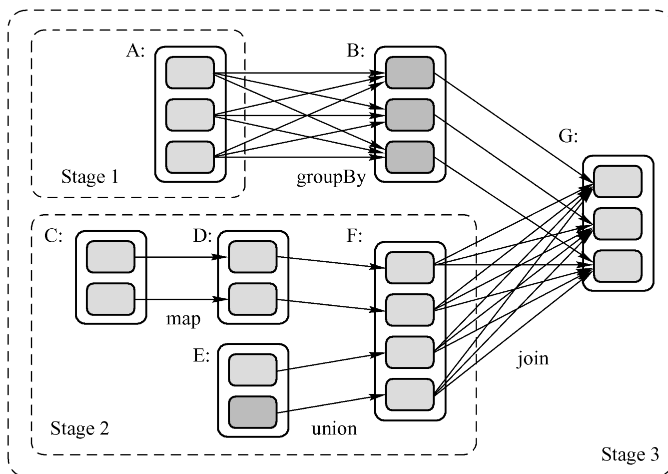


图 4-9　DAG 划分 Stage 及 Stage 并行计算示意图

其中，实线圆角方框标识的是 RDD，方框中的矩形块为 RDD 的分区。

在图 4-9 中，RDD A 到 RDD B 之间，以及 RDD F 到 RDD G 之间的数据需要经过 Shuffle 过程，因此 RDD A 和 RDD F 分别是 Stage 1 跟 Stage 3 和 Stage 2 跟 Stage 3 的划分点。而 RDD B 到 RDD G 之间，以及 RDD C 到 RDD D 到 RDD F 和 RDD E 到 RDD F 之间的数据不

需要经过 Shuffle 过程，因此，RDD G 和 RDD B 的依赖是窄依赖，RDD B 和 RDD G 划分到同一个 Stage 3，RDD F 和 RDD D 加 RDD E 的依赖，以及 RDD D 和 RDD C 的依赖是窄依赖，RDD C、RDD D、RDD E 和 RDD F 划分到同一个 Stage 2。Stage 1 和 Stage 2 是相互独立的，可以并发执行。而由于 Stage 3 依赖 Stage 1 和 Stage 2 的计算结果，所以 Stage 3 最后执行计算。

根据以上 RDD 依赖关系的描述，在图 4-9 中的操作算子中，map 和 union 是窄依赖的操作，因为子 RDD（如 D）的分区只依赖父 RDD（如 C）的一个分区，其他常见的窄依赖的操作还有 filter、flatMap 和 join（每个分区和已知的分区 join）等。groupByKey 和 join 是宽依赖操作，其他常见的宽依赖操作还有 reduceByKey 等。

由此可见，在 DAGScheduler 的调度过程中，Stage 阶段的划分是根据是否有 Shuffle 过程，也就是当存在 ShuffleDependency 的宽依赖时，需要进行 Shuffle，此时才会将作业（Job）划分成多个 Stage。

4.4.4 DAGScheduer 划分 Stage 的具体算法

理解了前面 DAGScheduler 划分 Stage 的原理后，现在结合 Spark 源代码来看看 Spark 是如何实现 DAGScheduler 划分 Stage 算法的。

RDD 的 Action 操作算子如 count 会触发一个 Spark Job，其实是 RDD 的 count 方法调用了 SparkContext 的 runJob 方法。然后，在 SparkContext 的 runJob 方法中调用 3 次重载的 runJob 方法。最后被调用的重载 runJob 方法中调用了 DAGScheduler 的 runJob 方法，从而进入 DAGScheduler 的源代码。

在 DAGScheduler 的源代码中，DAGScheduler 的 runJob 方法中调用 submitJob 方法。submitJob 方法中最重要的是创建一个 JobWaiter 对象，以及创建 JobSubmitted 事件对象把 JobWaiter 对象发送给 DAGScheduler 的内嵌类 DAGSchedulerEventProcessLoop 对象实例。DAGSchedulerEventProcessLoop 在 doOnReceive 方法中反过来调用了 DAGScheduler 中实现划分 Stage 算法很关键的 handleJobSubmitted 方法。

DAGScheduler 的 handleJobSubmitted 方法中调用 newResultStage 方法，newResultStage 方法根据 finalRDD 创建 finalStage，这时便真正开始了 Stage 的划分。DAGScheduler 的 handleJobSubmitted 方法中最后调用 submitStage 方法，根据 RDD 的依赖关系，递归提交所有的 Stage。

DAGScheduler 的 handleJobSubmitted 方法关键的代码如下。

```
private[scheduler] def handleJobSubmitted(jobId:Int,
   finalRDD:RDD[_],
   func:(TaskContext,Iterator[_]) => _,
   partitions:Array[Int],
   callSite:CallSite,
   listener:JobListener,
   properties:Properties) {
var finalStage:ResultStage = null
try {
```

```
        //新 Stage 的创建过程中可能会抛出异常,例如:job 运行在 HadoopRDD 之上,而 Hadoop
        底层的 HDFS 文件已经被删除了,就会抛出异常
        //以 finalRDD 为基础,开始整个 Job 的 Stage 划分
        finalStage = newResultStage(finalRDD,func,partitions,jobId,callSite)
    } catch {
        …
    }

    …
    //从 finalStage 开始,根据 RDD 的依赖关系,递归提交所有的 Stage
    submitStage(finalStage)
    …
}
```

DAGScheduler 的 newResultStage 方法中调用 getParentStagesAndId 方法得到 ResultStage 所有依赖的 Parent Stage 列表，以及 ResultStage 的 Stage ID。

DAGScheduler 的 newResultStage 方法的代码如下。

```
private def newResultStage(
      rdd:RDD[_],
      func:(TaskContext,Iterator[_]) => _,
      partitions:Array[Int],
      jobId:Int,
      callSite:CallSite):ResultStage = {
   //根据当前的 RDD 和 Job ID 得到所有依赖的 Parent Stage 列表,以及 ResultStage ID
   val (parentStages:List[Stage],id:Int) = getParentStagesAndId(rdd,jobId)
   //创建 ResultStage 对象实例
   val stage = new ResultStage(id,rdd,func,partitions,parentStages,jobId,callSite)
   stageIdToStage(id) = stage
   updateJobIdStageIdMaps(jobId,stage)
   stage
 }
```

DAGScheduler 的 getParentStagesAndId 方法比较简单，主要是调用 getParentStages 方法，其他就是得到一个当前最大的 Stage ID。

DAGScheduler 的 getParentStagesAndId 方法的代码如下。

```
private def getParentStagesAndId(rdd:RDD[_],firstJobId:Int):(List[Stage],Int) = {
      val parentStages = getParentStages(rdd,firstJobId)
      val id = nextStageId.getAndIncrement()
      (parentStages,id)
   }
```

可以说，DAGScheduler 的 getParentStages 方法是划分 Stage 的核心实现。这个方法的输出是获取或创建给定的 RDD 在 Job 中的所有 Parent Stage 列表。在该方法中，Stage 的划分由最后一个 Result Stage 开始，从后往前回溯边划分边创建。给定的 RDD 会先被压入 Stack，以便被逐个访问。对每个被访问的 RDD，判断其 Dependency 是否是 ShuffleDependency 宽依赖：如果是，就调用 getShuffleMapStage 方法获取或创建 Stage，并放到一个临时 HashSet 中，便于统一最后转为列表返回；如果不是，就把 Dependency 的 RDD 也压入 Stack 中，继续判断该 RDD 的 Dependency。方法的结束标志是 Stack 中所有压入的临时 RDD 都被访问过。

DAGScheduler 的 getParentStages 方法的代码如下。

```
private def getParentStages(rdd:RDD[_],firstJobId:Int):List[Stage] = {
  val parents = new HashSet[Stage]
  val visited = new HashSet[RDD[_]] //存储已经被访问的 RDD,构建时是从后往前回溯的
  //维护一个栈,防止由于递归访问引起的栈道出异常(StackOverflowError)
  val waitingForVisit = new Stack[RDD[_]] //存储需要被处理的 RDD
  def visit(r:RDD[_]) {
    if (!visited(r)) {
      visited += r//如果 RDD 没被访问过,那么这次就将此 RDD 加入访问过的 RDD HashSet 中
      //由于 Partition 未知,所以需要注册 RDD
      for (dep <- r. dependencies) {
        dep match {
          //RDD 的依赖是 ShuffleDependency 宽依赖,获取或创建新 Stage
          case shufDep:ShuffleDependency[_,_,_] =>
            parents += getShuffleMapStage(shufDep,firstJobId)
          //RDD 的依赖不是 ShuffleDependency 宽依赖,把依赖的 RDD 压入待访问 RDD 的 Stack 中
          case _ =>
            waitingForVisit. push(dep. rdd)
        }
      }
    }
  }
  //先压入给定的 RDD 到待访问 Stack 中
  waitingForVisit. push(rdd)
  while (waitingForVisit. nonEmpty) { //如果 Stack 不为空,继续访问 Stack 的下一个 RDD
    visit(waitingForVisit. pop())
  }
  parents. toList
}
```

剩下的过程就比较容易理解了，DAGScheduler 的 getShuffleMapStage 方法先根据给定的 ShuffleDependency 的 shuffleId 在 shuffleToMapStage 这个 HaskMap 中获取相应的 ShuffleMapStage，若没有找到，就创建一个新的 Stage 返回。

```
private def getShuffleMapStage(
        shuffleDep: ShuffleDependency[_,_,_],
        firstJobId: Int): ShuffleMapStage = {
    shuffleToMapStage.get(shuffleDep.shuffleId) match {
        //找到 Stage 则直接返回
        case Some(stage) => stage
        //否则，获取所有没有生成 Stage 的 ShuffleDependency，并以此生成 Stage
        case None =>
          //获取注册的 shuffle 父依赖
        getAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>
          shuffleToMapStage(dep.shuffleId) = newOrUsedShuffleStage(dep, firstJobId)
        }
         //注册当前的 shuffleDep 依赖
        val stage = newOrUsedShuffleStage(shuffleDep, firstJobId)
        shuffleToMapStage(shuffleDep.shuffleId) = stage
        stage
    }
}
```

DAGScheduler 的 getAncestorShuffleDependencies 方法的核心逻辑是返回在 shuffleToMapStage HashMap 中不存在对应的 Stage 的 ShuffleDependency，代码如下。

```
private def getAncestorShuffleDependencies(rdd: RDD[_]): Stack[ShuffleDependency[_,_,_]] = {
    val parents = new Stack[ShuffleDependency[_,_,_]]
    val visited = newHashSet[RDD[_]]
    //维护一个栈，防止由于递归访问引起的栈溢出异常(StackOverflowError)
    val waitingForVisit = new Stack[RDD[_]]
    def visit(r: RDD[_]) {
      if (!visited(r)) {
        visited += r
       for(dep <- r.dependencies) {
            dep match {
              caseshufDep: ShuffleDependency[_,_,_] =>
                //如果该 ShuffleDependency 没有生成过 Stage
                //则压入 parents Stack 等待返回
                if (!shuffleToMapStage.contains(shufDep.shuffleId)) {
                  parents.push(shufDep)
                }
              case _ =>
            }
            waitingForVisit.push(dep.rdd)
```



```
              }
          }
      }

      waitingForVisit.push(rdd)
      while (waitingForVisit.nonEmpty) {
          visit(waitingForVisit.pop())
      }
      parents
  }
```

DAGScheduler 的 newOrUsedShuffleStage 方法的代码如下。

```
private def newOrUsedShuffleStage(
        shuffleDep:ShuffleDependency[_,_,_],
        firstJobId:Int):ShuffleMapStage = {
    valrdd = shuffleDep.rdd
    valnumTasks = rdd.partitions.length      //获取任务数目
    val stage = newShuffleMapStage(rdd,numTasks,shuffleDep,firstJobId,rdd.creationSite)   //新建 shuffleMapStage
    if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {
        valserLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)
        vallocs = MapOutputTracker.deserializeMapStatuses(serLocs)
        (0 untillocs.length).foreach { i =>
            if (locs(i) ne null) {
                //locs(i) will be null if missing
                stage.addOutputLoc(i,locs(i))
            }
        }
    } else {
        因为 Partition 未知,无法构造 RDD,需要在 mapOutputTracker 及缓存中注册 RDD
        logInfo("Registering RDD " + rdd.id + " (" + rdd.getCreationSite + ")")
        mapOutputTracker.registerShuffle(shuffleDep.shuffleId,rdd.partitions.length)
    }
    stage
}
```

至此，整个 DAGScheduler 划分 Stage 的过程已经介绍完毕。Stage 划分完成后，DAGScheduler 就会回到 HandleJobSubmitted 方法中调用 submitStage 方法。在 submitStage 方法中，从 finalStage（ResultStage 对象实例）开始回溯，直到没有 Parent Stage 为止，提交整个 Job 的所有 Stage，在某一个 Stage，DAGScheduler 会调用 submitMissingTasks 方法把 Tasks 提交给 TaskScheduler 进行细粒度的 Task 调度。

下面介绍在 Stage 内部 Task 获取最佳位置的算法。

4.4.5 Stage 内部 Task 获取最佳位置的算法

在 DAGScheudler 的 submitMissingTasks 方法中体现了利用 RDD 的本地性来得到 Task 的本地性，从而获取 Stage 内部 Task 的最佳位置。DAGScheudler 的 submitMissingTasks 方法会通过调用 getPreferredLocs 方法得到 Task 最佳位置，并把结果存放到 taskIdToLocations Map 中以便使用。

DAGScheudler 的 submitMissingTasks 方法的相关部分关键代码如下：

```
private def submitMissingTasks(stage:Stage,jobId:Int) {
  …

  //首先标识出将要计算分区的索引
  val partitionsToCompute:Seq[Int] = stage.findMissingPartitions()

  …

  val taskIdToLocations :Map[Int,Seq[TaskLocation]] = try {
    stage match {
      case s:ShuffleMapStage =>
        partitionsToCompute.map { id => (id,getPreferredLocs(stage.rdd,id))}.toMap
      case s:ResultStage =>
        val job = s.activeJob.get
        partitionsToCompute.map { id =>
          val p = s.partitions(id)
          (id,getPreferredLocs(stage.rdd,p))
        }.toMap
    }
  …

  stage.makeNewStageAttempt(partitionsToCompute.size,taskIdToLocations.values.toSeq)
  …

  val tasks:Seq[Task[_]] = try {
    stage match {
      case stage:ShuffleMapStage =>
        partitionsToCompute.map { id =>
          val locs = taskIdToLocations(id)
          val part = stage.rdd.partitions(id)
          new ShuffleMapTask(stage.id,stage.latestInfo.attemptId,
            taskBinary,part,locs,stage.internalAccumulators)
        }
```



```

            case stage: ResultStage =>
              val job = stage.activeJob.get
              partitionsToCompute.map { id =>
                val p: Int = stage.partitions(id)
                val part = stage.rdd.partitions(p)
                val locs = taskIdToLocations(id)
                new ResultTask(stage.id, stage.latestInfo.attemptId,
                  taskBinary, part, locs, id, stage.internalAccumulators)
              }
          }
      }
    …
  }
```

DAGScheudler 的 getPreferredLocs 方法只是调用 getPreferredLocsInternal 方法。

DAGScheudler 的 getPreferredLocs 方法代码如下。

```
private[spark]
def getPreferredLocs(rdd: RDD[_], partition: Int): Seq[TaskLocation] = {
  getPreferredLocsInternal(rdd, partition, new HashSet)
}
```

DAGScheudler 的 getPreferredLocsInternal 方法具体实现了一个 Partition 的数据本地性的算法。在具体算法实现时，首先查询 DAGScheduler 的内存数据结构中是否存在当前 Paritition 的数据本地性的信息，如果有则直接返回，如果没有首先会调用 rdd.getPreferedLocations，例如，想让 Spark 运行在 HBase 上或一种现在还没有直接支持的数据库上面，此时开发者需要自定义 RDD，为了保证 Task 计算的数据本地性，最为关键的方式是必须实现 RDD 的 getPreferedLocations，数据本地性在底层运行之前就完成了。

DAGScheudler 的 getPreferredLocsInternal 方法代码如下：

```
private def getPreferredLocsInternal(
    rdd: RDD[_],
    partition: Int,
    visited: HashSet[(RDD[_], Int)]): Seq[TaskLocation] = {
  // 如果 RDD 分区在之前的迭代中已经被访问过,就没必要重复访问
  if (!visited.add((rdd, partition))) {
    //如果分区之前已经询问过了,就返回 Nil
    return Nil
  }
  //首先,优先获取缓存中的 RDD 分区地址
   val cached = getCacheLocs(rdd)(partition)
    if (cached.nonEmpty) {
```

```
        return cached
    }
    //然后,如果缓存中没有 RDD 分区地址,则判断 RDD 本身是否有首选的地址。若有,就用来构造 TaskLocation 并返回
    val rddPrefs = rdd.preferredLocations(rdd.partitions(partition)).toList
    if (rddPrefs.nonEmpty) {
        return rddPrefs.map(TaskLocation(_))
    }
    //最后,如果 RDD 有窄依赖,就挑选第一个窄依赖的第一个分区的首选地址
    rdd.dependencies.foreach {
        case n:NarrowDependency[_] =>
          for (inPart <- n.getParents(partition)) {
            val locs = getPreferredLocsInternal(n.rdd, inPart, visited)
            if (locs != Nil) {
              return locs
            }
          }

        case _ =>
    }

    Nil
  }
```

DAGScheduler 计算数据本地性时巧妙地借助了 RDD 自身的 getPreferedLocations 中的数据，最大化地优化效率，因为 getPreferedLocations 中表明了每个 Partition 的数据本地性，虽然当前 Partition 可能被 persist 或者 checkpoint，但是 persist 或者 checkpoint 默认情况下肯定是和 getPreferedLocations 中的 Partition 数据本地性一致，所以这就极大地简化了 Task 数据本地性算法的实现和效率的优化。

注意，数据本地性是指：确定数据在哪个结点上，就到哪个结点的 Executor 上去运行。

Section 4.5 底层的 Task 调度器（TaskScheduler）

TaskScheduler 的核心任务是提交 TaskSet 到集群运算并汇报结果。

1）为 TaskSet 创建和维护一个 TaskSetManager，并追踪任务的本地性及错误信息。

2）遇到 Straggle 任务会放到其他结点进行重试。

3）向 DAGScheduler 汇报执行情况，包括在 Shuffle 输出丢失时报告 fetch failed 错误等信息。

4.5.1 TaskScheduer 原理剖析

通过之前 DAGScheduler 的介绍可以知道，DAGScheduler 将划分的一系列 Stage（每个 Stage 封装一个 TaskSet），按照 Stage 的先后顺序依次提交给底层的 TaskScheduler 去执行。下面来分析 TaskScheduler 接收到 DAGScheduler 的 Stage 任务后，是如何管理 Stage（TaskSet）的生命周期的。

首先，回顾一下 DAGScheduler 在 SparkContext 中实例化时，TaskScheduler 和 SchedulerBackend 就已经先在 SparkContext 的 createTaskScheduler 创建出实例对象了。

📖 注意，虽然 Spark 支持多种资源部署模式（包括 Local、Standalone、YARN 和 Mesos 等），但是底层调度器 TaskScheduler 接口的实现类都是 TaskSchedulerImpl。并且，为了方便读者理解 TaskScheduler，对于 SchedulerBackend 的实现也只专注 Standalone 部署模式下的具体实现 SparkDeploySchedulerBackend 来做分析。

TaskSchedulerImpl 在 createTaskScheduler 方法中实例化后，就立即调用自己的 initialize 方法把 SparkDeploySchedulerBackend 的实例对象传进来，从而赋值给 TaskSchedulerImpl 的 backend。在 TaskSchedulerImpl 的 initialize 方法中，根据调度模式的配置创建实现了 SchedulerBuilder 接口的相应实例对象，并且创建的对象会立即调用 buildPools 创建相应数量的 Pool 存放和管理 TaskSetManager 的实例对象。实现 SchedulerBuilder 接口的具体类都是 SchedulerBuilder 的内部类。

1）FIFOSchedulableBuilder：调度模式是 SchedulingMode. FIFO，使用先进先出策略调度。这是默认模式，在该模式下，只有一个 TaskSetManager 池。

2）FairSchedulableBuilder：调度模式是 SchedulingMode. FAIR，使用公平策略调度。

在 createTaskScheduler 方法返回后，TaskSchedulerImpl 通过 DAGScheduler 的实例化过程设置 DAGScheduler 的实例对象，然后调用自己的 start 方法。在 TaskSchedulerImpl 调用 start 方法时，会调用 SparkDeploySchedulerBackend 的 start 方法，在 SparkDeploySchedulerBackend 的 start 方法中会最终注册应用程序 AppClient。TaskSchedulerImpl 的 start 方法中还会根据配置判断是否周期性地检查任务的推测执行。

TaskSchedulerImpl 启动后，就可以接收 DAGScheduler 的 submitMissingTasks 方法提交过来的 TaskSet 进行进一步处理。TaskSchedulerImpl 在 submitTasks 中初始化一个 TaskSetManager 对其生命周期进行管理，当 TaskSchedulerImpl 得到 Worker 结点上的 Executor 计算资源时，会通过 TaskSetManager 来发送具体的 Task 到 Executor 上执行计算。

如果 Task 执行过程中有错误导致失败，会调用 TaskSetManager 来处理 Task 失败的情况，进而通知 DAGScheduler 结束当前的 Task。TaskSetManager 会将失败的 Task 再次添加到待执行的 Task 队列中。

📖 注意，Spark Task 允许失败的次数默认是 4 次，在 TaskSchedulerImpl 初始化时通过 spark. task. maxFailures 设置该默认值。

如果 Task 执行完毕，执行的结果会反馈给 TaskSetManager，由 TaskSetManager 通知 DAGScheduler。DAGScheduler 根据是否还存在待执行的 Stage，继续迭代提交对应的 TaskSet 给 TaskScheduler 去执行，或者输出 Job 的结果。

结合第 5 章执行器（Executor）可知，通过下面的调度链，Executor 把 Task 执行的结果返回给调度器（Scheduler）。

1）Executor. run。

2）CoarseGrainedExecutorBackend. statusUpdate（发送 StatusUpdate 消息）。

3）CoarseGrainedSchedulerBackend. receive（处理 StatusUpdate 消息）。

4）TaskSchedulerImpl. statusUpdate。

5）TaskResultGetter. enqueueSuccessfulTask 或者 enqueueFailedTask。

6）TaskSchedulerImpl. handleSuccessfulTask 或者 handleFailedTask。

7）TaskSetManager. handleSuccessfulTask 或者 handleFailedTask。

8）DAGScheduler. taskEnded。

9）DAGScheduler. handleTaskCompletion。

上面的调度链值得关注的是：第 7）步中，TaskSetManager 的 handleFailedTask 方法会将失败的 Task 再次添加到待执行 Task 队列中。在第 6）步中，TaskSchedulerImpl 的 handleFailedTask 方法在 TaskSetManager 的 handleFailedTask 方法返回后，会调用 CoarseGrainedSchedulerBackend 的 reviveOffers 方法给重新执行的 Task 获取资源。

4.5.2 TaskScheduer 源代码解析

下面通过源代码解析来看一下 TaskScheduler 是如何调度和管理 TaskSet 的任务。

1. TaskScheduler 实例化源代码

TaskScheduler 和 DAGScheduler 都在 SparkContext 实例化的时候一同实例化。

SparkContext 源代码中与 TaskScheduler 实例化相关的代码如下。

```
…
@volatile private var _taskScheduler:TaskScheduler = _

…
  private[spark] def taskScheduler:TaskScheduler = _taskScheduler
  private[spark] def taskScheduler_ = (ts:TaskScheduler):Unit = {
    _taskScheduler = ts
  }
…
 //实例化 SchedulerBackend 对象和 TaskScheduler 对象
 val (sched,ts) = SparkContext. createTaskScheduler(this,master)
 _schedulerBackend = sched
 _taskScheduler = ts

 //实例化 DAGScheduler 对象
```

```
_dagScheduler = new DAGScheduler(this)

//启动 TaskScheduler 实例对象,等待 DAGScheduler 提交 TaskSet 给 TaskScheduler
_taskScheduler.start()
...
```



SparkContext 的 createTaskScheduler 方法中与 Standalone 部署模式相关的代码如下。

```
/**
  * Create a task scheduler based on a given master URL.
  * Return a 2-tuple of the scheduler backend and the task scheduler.
  */
private def createTaskScheduler(
      sc:SparkContext,
      master:String):(SchedulerBackend,TaskScheduler) = {
...

    master match {
  ...
        //Spark Standalone 部署模式下 TaskScheduler 和 SchedulerBackend 分别由各自对应
        //的实现类 TaskSchedulerImpl 和 SparkDeploySchedulerBackend 来实例化对象
        case SPARK_REGEX(sparkUrl) =>
          val scheduler = new TaskSchedulerImpl(sc)
          val masterUrls = sparkUrl.split(",").map("spark://" + _)
          val backend = new SparkDeploySchedulerBackend(scheduler,sc,masterUrls)
          scheduler.initialize(backend)
          (backend,scheduler)

  ...
    }
  }
  }
```

2. TaskScheduler 初始化源代码

TaskScheduler（实际上在实现类 TaskSchedulerImpl 的 initialize 方法中）在初始化的过程中设置对 SchedulerBackend 对象的引用，实例化 SchedulerBuilder 具体实现类的对象用来创建和管理 TaskSetManager 池。

TaskSchedulerImpl 源代码中的相关代码如下。

```
//默认的调度模式是先进先出(FIFO)
private val schedulingModeConf = conf.get("spark.scheduler.mode","FIFO")
val schedulingMode:SchedulingMode = try {
  SchedulingMode.withName(schedulingModeConf.toUpperCase)
```

```
    } catch {
  …}
    def initialize(backend:SchedulerBackend) {
      //设置对 SchedulerBackend 对象的引用
      this.backend = backend
      //rootpoot 的名字暂时设置为空值
      rootPool = new Pool("",schedulingMode,0,0)
      schedulableBuilder = {
        schedulingMode match {
          //调度模式是 FIFO
          case SchedulingMode.FIFO  =>
            //FIFOSchedulableBuilder 的 rootPool 里面直接添加 TaskSetManager
            new FIFOSchedulableBuilder(rootPool)
          //调度模式是 FAIR
          case SchedulingMode.FAIR  =>
            //FairSchedulableBuilder 的 rootPool 根据配置文件可以挂若干个子 pool,
            //每个 pool 里面都添加 TaskSetManager
            new FairSchedulableBuilder(rootPool,conf)
        }
      }
     //构建 TaskSetManager 池
     schedulableBuilder.buildPools()
   }
```

3. TaskScheduler 启动源代码

TaskScheduler 实例对象在 DAGScheduler 实例化之后启动，并且 TaskScheduler 启动的过程由 TaskSchedulerImpl 具体实现。在启动过程中，主要是调用 SchedulerBackend 的启动方法，然后对不是本地部署模式并且开启任务的推测执行（设置 spark.speculation 为 true）情况，根据配置判断是否周期性地调用 TaskSetManager 的 checkSpeculatableTasks 方法检查任务的推测执行。SparkDeploySchedulerBackend 的 start 方法中会最终注册应用程序 AppClient。关于 SchedulerBackend 的更多源代码解析请参考 4.6 节。

TaskSchedulerImpl 源代码中的 start 方法的相关代码如下。

```
override def start() {
  //Spark Standalone 部署模式下调用 SparkDeploySchedulerBackend 的 start 方法,
  //SparkDeploySchedulerBackend 的 start 方法中会最终注册应用程序 AppClient
  backend.start()

  //不是本地部署模式并且开启了任务的推测执行
  if (!isLocal && conf.getBoolean("spark.speculation",false)) {
    logInfo("Starting speculative execution thread")
    speculationScheduler.scheduleAtFixedRate(new Runnable {
```

```
            override def run():Unit  = Utils.tryOrStopSparkContext(sc) {
              //最终会调用调度池中的 TaskSetManager 的 checkSpeculatableTasks 方法
              //来检查推测执行的任务
              checkSpeculatableTasks()
            }
              }, SPECULATION_INTERVAL_MS, SPECULATION_INTERVAL_MS,
TimeUnit.MILLISECONDS)
        }
    }
```



4. TaskScheduler 提交任务源代码

TaskSchedulerImpl 启动后，就可以接收 DAGScheduler 的 submitMissingTasks 方法提交过来的 TaskSet 进行进一步处理了。对于 ShuffleMapStage 类型的 Stage，DAGScheduler 初始化一组 ShuffleMapTask 实例对象；对于 ResultStage 类型的 Stage，DAGScheduler 初始化一组 ResultTask 实例对象。最后，DAGScheduler 将这组 ResultTask 实例对象封装成 TaskSet 实例对象提交给 TaskSchedulerImpl。

注意，ShuffleMapTask 是根据 Stage 所依赖的 RDD 的 partition 分布产生跟 partition 数量相等的 Task，这些 Task 根据 partition 的本地性分布在不同的集群结点；ResultTask 负责输出整个 Job 的结果。

DAGScheduler 的 submitMissingTasks 方法的部分关键代码如下。

```
  /** Called when stage's parents are available and we can now do its task.  */
  private def submitMissingTasks(stage:Stage,jobId:Int) {
    logDebug("submitMissingTasks(" + stage + ")")
...

    val tasks:Seq[Task[_]] = try {
      stage match {
        //对于 ShuffleMapStage,初始化一组 ShuffleMapTask 实例对象



          case stage:ShuffleMapStage =>
            partitionsToCompute.map { id =>
            vallocs = taskIdToLocations(id)
            val part = stage.rdd.partitions(id)
            new ShuffleMapTask (stage.id,stage.latestInfo.attemptId,
              taskBinary,part,locs,stage.internalAccumulators)
          }
        //对于 ResultStage,初始化一组 ResultTask 实例对象

        case stage:ResultStage =>
```

```
              val job = stage.activeJob.get
              partitionsToCompute.map { id =>
                val p:Int = stage.partitions(id)
                val part = stage.rdd.partitions(p)
                val locs = taskIdToLocations(id)
                new ResultTask(stage.id,stage.latestInfo.attemptId,
                  taskBinary,part,locs,id,stage.internalAccumulators)
              }
          }
      } catch {
    …

      if (tasks.size > 0) {
        logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" + stage.rdd + ")")
        stage.pendingPartitions ++= tasks.map(_.partitionId)
        logDebug("New pending partitions:" + stage.pendingPartitions)
        //将生成的所有 Task 封装成一个 TaskSet,提交给 TaskScheduler 的 submitTasks
        //实际上调用 TaskSchedulerImpl 的 submitTasks 方法进一步处理
        taskScheduler.submitTasks(new TaskSet(
          tasks.toArray,stage.id,stage.latestInfo.attemptId,jobId,properties))
         stage.latestInfo.submissionTime = Some(clock.getTimeMillis())
      } else {
    …
      }
    }
}
```

TaskSchedulerImpl 在 submitTasks 中初始化一个 TaskSetManager，并通过 SchedulerBuilder 对其生命周期进行管理，最后调用 SchedulerBackend 的 reviveOffers 方法进行 TaskSet 所需资源的分配。在 TaskSet 得到足够的资源后，在 SchedulerBackend 的 launchTasks 方法中将 TaskSet 中的 Task 一个一个地发送到 Executor 去执行。

TaskSchedulerImpl 的 submitTasks 方法的部分关键代码如下。

```
override def submitTasks(taskSet:TaskSet) {
  val tasks  = taskSet.tasks
  logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
    val manager = createTaskSetManager(taskSet,maxTaskFailures)
     val stage  = taskSet.stageId
    val stageTaskSets  =
      taskSetsByStageIdAndAttempt.getOrElseUpdate(stage,new HashMap[Int,TaskSetManager])
      stageTaskSets(taskSet.stageAttemptId) = manager
```

```
        val conflictingTaskSet = stageTaskSets.exists { case (_,ts) =>
          ts.taskSet != taskSet && ! ts.isZombie
        }
    ...
      //SchedulerBuilder 将新建的 TaskSetManager 实例对象添加到关联的 Pool 中
      schedulableBuilder.addTaskSetManager(manager,manager.taskSet.properties)

      //对于非本地部署模式,如果没有接收到 Task,就周期性地警告或者取消 Task
      if (! isLocal && ! hasReceivedTask) {
        starvationTimer.scheduleAtFixedRate(new TimerTask() {
          override def run() {
            if (! hasLaunchedTask) {
              logWarning("Initial job has not accepted any resources;" +
                "check your cluster UI to ensure that workers are registered " +
                "and have sufficient resources")
            } else {
              this.cancel()
            }
          }
        },STARVATION_TIMEOUT_MS,STARVATION_TIMEOUT_MS)
      }
      hasReceivedTask = true
    }
    //Spark Standalone 部署模式调用 SparkDeploySchedulerBackend 的 reviveOffers 方法
    //进行 TaskSet 所需资源的分配,在得到足够的资源后,将 TaskSet 中的 Task
    //一个一个地发送到 Executor 去执行
    backend.reviveOffers()
  }
```

接下来继续讲解 SchedulerBackend 的原理剖析和源代码解析。

Section 4.6 调度器的通信终端（SchedulerBackend）

4.6.1 SchedulerBackend 原理

以 Spark Standalone 部署方式为例，SparkDeploySchedulerBackend 在启动时构造了 AppClient 实例，并在该实例 start 时启动了 ClientEndpoint 消息循环体，ClientEndpoint 在启动时会向 Master 注册当前程序。而 SparkDeploySchedulerBackend 的父类 CoarseGrainedSchedulerBackend 在 start 时会实例化类型为 DriverEndPoint（这就是程序运行时的经典对象 Driver）的

消息循环体，SparkDeploySchedulerBackend 专门负责收集 Worker 上的资源信息，当 ExecutorBackend 启动时会发送 RegisteredExecutor 信息向 DriverEndpoint 注册，此时 SparkDeploySchedulerBackend 就掌握了当前应用程序拥有的计算资源，TaskScheduler 就是通过 SparkDeploySchedulerBackend 拥有的计算资源来具体运行 Task 的。

4.6.2 SchedulerBackend 源代码解析

下面通过源代码解析来看一下 Spark Standalone 部署方式下的 SparkDeploySchedulerBackend 是如何收集和分配资源给调度的 Task 用的。

SparkDeploySchedulerBackend 的 start 方法的相关代码如下。

```
override def start() {
  //先调用父类 CoarseGrainedSchedulerBackend 的 start 方法
  super.start()
  launcherBackend.connect()

  //Executor 的 endpoint 循环消息体进行汇报
  val driverUrl = rpcEnv.uriOf(SparkEnv.driverActorSystemName,
    RpcAddress(sc.conf.get("spark.driver.host"), sc.conf.get("spark.driver.port").toInt),
    CoarseGrainedSchedulerBackend.ENDPOINT_NAME)
  val args = Seq(
      "--driver-url", driverUrl,
      "--Executor-id", "{{EXECUTOR_ID}}",
      "--hostname", "{{HOSTNAME}}",
      "--cores", "{{CORES}}",
      "--app-id", "{{APP_ID}}",
      "--worker-url", "{{WORKER_URL}}")
  val extraJavaOpts = sc.conf.getOption("spark.Executor.extraJavaOptions")
      .map(Utils.splitCommandString).getOrElse(Seq.empty)
  val classPathEntries = sc.conf.getOption("spark.Executor.extraClassPath")
      .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)
  val libraryPathEntries = sc.conf.getOption("spark.Executor.extraLibraryPath")
      .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)
 …

  //Executor 注册时需提供的必要的一些配置信息
  val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
  val javaOpts = sparkJavaOpts ++ extraJavaOpts
  //指定了具体为当前应用程序启动 Executor 进程的入口类
  //为 CoarseGrainedExecutorBackend
   val command = Command("org.apache.spark.Executor.CoarseGrainedExecutorBackend",
args, sc.ExecutorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)
```

```
    val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
    val coresPerExecutor = conf.getOption("spark.Executor.cores").map(_.toInt)
    //将当前 Application 的所有相关信息分装在 ApplicationDescription 实例对象中
    val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.ExecutorMemory, command, appUIAddress, sc.eventLogDir, sc.eventLogCodec, coresPerExecutor)
    //创建 AppClient 实例对象,用于 Application 跟 Spark 部署的集群通信
    client = newAppClient(sc.env.rpcEnv, masters, appDesc, this, conf)
    // 启动 AppClient,其方法内部会创建其内部类 ClientEndpoint 的实例对象
    client.start()
    launcherBackend.setState(SparkAppHandle.State.SUBMITTED)
    waitForRegistration()
    launcherBackend.setState(SparkAppHandle.State.RUNNING)
  }
```

SparkDeploySchedulerBackend 的父类 CoarseGrainedSchedulerBackend 的 start 方法和相关代码如下。

```
  override def start() {
    val properties = newArrayBuffer[(String, String)]
    for ((key, value) <- scheduler.sc.conf.getAll) {
      if (key.startsWith("spark.")) {
        properties += ((key, value))
      }
    }

    //创建 CoarseGrainedSchedulerBackend 内部类 DriverEndpoint 的实例对象作为消息
    //循环体,以便处理接收的主要消息 RegisterExecutor(这个消息来自 ExecutorBackend)
     driverEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME, createDriverEndpoint(properties))
  }

 private[spark] object CoarseGrainedSchedulerBackend {
    val ENDPOINT_NAME = "CoarseGrainedScheduler"
  }
```

AppClient 的 start 方法的代码如下。

```
  def start() {
    //Just launch an rpcEndpoint; it will call back into the listener.
    //创建 AppClient 内部类 ClientEndpoint 的实例对象作为消息循环体,
    //以便向 Master 注册当前的 Application
    endpoint.set(rpcEnv.setupEndpoint("AppClient", new ClientEndpoint(rpcEnv)))
  }
```



接下来继续讲解 Spark 程序的的注册机制。

4.6.3 Spark 程序的注册机制

在上面的源代码分析中，AppClient 在启动时创建了 AppClient 内部类 ClientEndpoint 的实例对象作为消息循环体，以便向 Master 注册当前的 Application。既然 ClientEndpoint 是 RpcEndpoint 的子类，那么就会有这样的生命周期：constructor→onStart→receive→onStop。根据这个原理，来看一下 ClientEndpoint 的 onStart 方法的代码。

```
override def onStart():Unit = {
    try {
      //向 Master 注册 Application,参数 1 代表第 1 次尝试注册
      registerWithMaster(1)
    } catch {
      case e:Exception =>
        logWarning("Failed to connect to master",e)
        markDisconnected()
        stop()
    }
  }
```

ClientEndpoint 在启动时就立即调用 registerWithMaster 来注册 Application，继续查看 registerWithMaster 方法的代码。

```
private def registerWithMaster(nthRetry:Int) {
  //向所有 Master 异步地尝试注册 Application
  registerMasterFutures.set(tryRegisterAllMasters())
  registrationRetryTimer.set(registrationRetryThread.scheduleAtFixedRate(new Runnable {
    override def run():Unit = {
      Utils.tryOrExit {
      //如果已经注册过,那么就取消向其他 Master 的注册,并且关闭注册用的线程池
          if (registered.get) {
            registerMasterFutures.get.foreach(_.cancel(true))
            registerMasterThreadPool.shutdownNow()
          //如果注册尝试次数等于或者超过 3 次,就表明本次注册失败
          } else if (nthRetry >= REGISTRATION_RETRIES) {
           markDead("All masters are unresponsive! Giving up.")
          //否则继续尝试向 Master 注册 Application
          } else {
             registerMasterFutures.get.foreach(_.cancel(true))
             registerWithMaster(nthRetry + 1)
           }
```

```
            }
          }
        //每次注册过程的时间不能超过20s,否则注册失败
        }, REGISTRATION_TIMEOUT_SECONDS, REGISTRATION_TIMEOUT_SECONDS, TimeUnit.SECONDS))
  }
```



ClientEndpoint 在 tryRegiesterAllMasters 方法中会向所有的 Master 尝试注册 Application。相关方法的代码如下。

```
private def tryRegisterAllMasters(): Array[JFuture[_]] = {
  for (masterAddress <- masterRpcAddresses) yield {
    registerMasterThreadPool.submit(new Runnable {
      override def run(): Unit = try {
        if (registered.get) {
          return
        }
        logInfo("Connecting to master " + masterAddress.toSparkURL + "...")
        valmasterRef =
          rpcEnv.setupEndpointRef(Master.SYSTEM_NAME, masterAddress, Master.ENDPOINT_NAME)
          //给 Master 发送 RegisterApplication 消息处理 Application 的注册
          masterRef.send(RegisterApplication(appDescription, self))
      } catch {
          case ie: InterruptedException => // Cancelled
          case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress", e)
      }
    })
  }
}
```

Master 也是 RpcEndpoint 的子类，所以可以通过 receive 方法接收 DeployMessage 类型的消息 RegisterApplication。部分相关代码如下。

```
  override def receive: PartialFunction[Any, Unit] = {
…

  case RegisterApplication(description, driver) => {
    //忽略 STANDBY 状态的 Master
    if (state == RecoveryState.STANDBY) {
      // ignore, don't send response
    } else {
```




```
            logInfo("Registering app " + description.name)
            //创建 ApplicationInfo 实例对象,封装诸如系统时间戳、Application ID、默认 Application 需要的最大 CPU Core 数量和 ClientEndpoint 的引用等信息
            val app = createApplication(description, driver)
             //具体注册 Application 到 Master 对象的内存结构中
             registerApplication(app)
             logInfo("Registered app " + description.name + " with ID " + app.id)
             //将新创建的 Application 持久化,以便错误能恢复
             persistenceEngine.addApplication(app)
             //返回 AppClient.ClientEndpoint 的 receive 方法处理 RegisteredApplication 消息
             driver.send(RegisteredApplication(app.id, self))
             //给注册好的 Application 分配 Executor 资源
               schedule()
          }
        }

  private def registerApplication(app: ApplicationInfo): Unit = {
    val appAddress = app.driver.address
    //忽略相同地址的 Application 的注册
    if (addressToApp.contains(appAddress)) {
      logInfo("Attempted to re-register application at same address:" + appAddress)
      return
    }

    applicationMetricsSystem.registerSource(app.appSource)
    //将注册好的 Application 加入到 Master 维护的关于 Application 的成员变量中
    apps += app
    idToApp(app.id) = app
    endpointToApp(app.driver) = app
    addressToApp(appAddress) = app
    waitingApps += app
  }
}
```

ClientEndpoint 最后在 receive 方法中得到来自 Master 注册好 Application 的确认。部分相关代码如下。

```
override def receive: PartialFunction[Any, Unit] = {
  case RegisteredApplication(appId_, masterRef) =>
…
  //保存 Master 返回的 Application ID
  appId.set(appId_)
```

```
    //标记 Application 注册完毕
    registered.set(true)
    //保存 Master 的通信端的引用
    master = Some(masterRef)
    listener.connected(appId.get)
  …
  }
```



至此，Application 向 Master 注册完毕。在上面的 RegisterApplication 中调用了 schedule 方法，这个方法将完成 Application 的调度，并在 Worker 结点上启动分配好的 Executor 给 Application 使用。关于具体如何把 Executor 分配给 Application，请参考第 5 章执行器（Executor）的解析。

4.6.4 Spark 程序对计算资源 Executor 的管理

从 TaskSchedulerImpl 的 submitTasks 的方法中可以知道，Spark Standalone 部署模式调用 SparkDeploySchedulerBackend 的 reviveOffers 方法进行 TaskSet 所需资源的分配，在得到足够的资源后，将 TaskSet 中的 Task 一个一个地发送到 Executor 去执行。下面来看一下这里的资源即 Executor 是如何得到和分配的。

SparkDeploySchedulerBackend 的 reviveOffers 方法很简单，就是发送一个 ReviveOffers 消息给内部类 DriverEndpoint，代码如下。

```
override def reviveOffers() {
    // DriverEndpoint 是 RpcEndpoint 的子类,会在 receive 方法里处理 ReviveOffers 消息
    driverEndpoint.send(ReviveOffers)
}
```

DriverEndpoint 的 receive 方法处理 ReviveOffers 消息也很简单，就是调用 makeOffers 方法，receive 方法的部分关键代码如下。

```
    override def receive:PartialFunction[Any,Unit] = {
…
        caseReviveOffers  =>
            //回调 DriverEndpoint 的 makeOffers 方法
            makeOffers()
…
    }
```

DriverEndpoint 的 makeOffers 方法首先过滤出 Alive 状态的 Executor 放到 activeExecutors HahMap 变量中，然后使用 id、ExecutorData.ExecutorHost 和 ExecutorData.freeCores 构建代表 Executor 可用资源的 WorkerOffer。最后是两个最重要的方法调用。先是调用 TaskSchedulerImpl 的 resourceOffers 得到 TaskDescription 的二维数组，包含 Task ID、Executor ID 和 Task In-

dex 等 Task 执行需要的信息。然后回调 DriverEndpoint 的 launchTask 给每个 Task 对应的 Executor 发送执行 Task 的 LaunchTask 消息（其实是由 CourseGrainedExecutorBackend 转发 LauchTask 消息）。DriverEndpoint 的 makeOffers 方法的代码如下。

```
private def makeOffers() {
    //只保留 Alive 状态的 Executor
    val activeExecutors = ExecutorDataMap.filterKeys(ExecutorIsAlive)
    //构建代表 Executor 可用资源的 WorkOffer
    val workOffers = activeExecutors.map { case (id,ExecutorData) =>
          newWorkerOffer(id,ExecutorData.ExecutorHost,ExecutorData.freeCores)
      }.toSeq
      //先调用 TaskSchedulerImpl 的 resourceOffers 得到 TaskDescripion 的二维数组
      //然后回调 lauchTask 给每个 Task 对应的 Executor 发送 LaunchTask 消息
     launchTasks(scheduler.resourceOffers(workOffers))
     }
```

TaskSchedulerImpl 的 resourceOffers 方法的代码如下。

```
def resourceOffers(offers:Seq[WorkerOffer]):Seq[Seq[TaskDescription]] = synchronized {
   // Mark each slave as alive and remember its hostname
   // Also track if new Executor is added
   var newExecAvail = false
   for (o <- offers) {
     ExecutorIdToHost(o.ExecutorId) = o.host
     ExecutorIdToTaskCount.getOrElseUpdate(o.ExecutorId,0)
     //如果在 ExecutorByHost 中没找到 o.host,则加入新的 Executor
     if (!ExecutorsByHost.contains(o.host)) {
         ExecutorsByHost(o.host) = new HashSet[String]()
         ExecutorAdded(o.ExecutorId,o.host)
         newExecAvail = true
       }
      for (rack <- getRackForHost(o.host)) {
         hostsByRack.getOrElseUpdate(rack,new HashSet[String]()) += o.host
       }
    }

    //为了避免将 Task 总是放在某些 Worker 上,这里随机打散这些 Task
    val shuffledOffers = Random.shuffle(offers)
    //构建分配好资源的 Task
    val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))
    val availableCpus = shuffledOffers.map(o => o.cores).toArray
    //得到调度模式排好序的 TaskSetManager
    val sortedTaskSets = rootPool.getSortedTaskSetQueue
```

```
    for (taskSet <- sortedTaskSets) {
      logDebug("parentName:%s,name:%s,runningTasks:%s".format(
        taskSet.parent.name,taskSet.name,taskSet.runningTasks))
      if (newExecAvail) {
        taskSet.ExecutorAdded()
      }
    }

    //按照调度模式排好的顺序获取 TaskSet,然后按照优先分配的就近原则分配资源
    //优先分配的就近原则依次是:PROCESS_LOCAL、NODE_LOCAL、NO_PREF、RACK_LOCAL 和 ANY
    var launchedTask = false
    for (taskSet <- sortedTaskSets;maxLocality <- taskSet.myLocalityLevels) {
      do {
        //为每个 TaskSet 分配 Executor 和 CPU cores 资源
        launchedTask = resourceOfferSingleTaskSet(
            taskSet,maxLocality,shuffledOffers,availableCpus,tasks)
      } while (launchedTask)
    }

    if (tasks.size > 0) {
      hasLaunchedTask = true
    }
    return tasks
  }
```

TaskSchedulerImpl 的 resourceOffers 方法返回二维数组 TaskDescription 后作为 DriverEndpoint 的 launchTasks 方法的参数。DriverEndpoint 的 launchTasks 方法中首先对传入的 tasks 进行扁平化操作（例如，将多维数组降维成一维数组），得到所有的 Task，然后遍历所有的 Task。在遍历过程中，调用 serialize( ) 方法对 Task 进行序列化，得到 serializedTask。判断如果 serialiedTask 大于等于 Akka 帧减去 Akka 预留空间大小，则调用 TaskSetManager 的 abort 方法终止该任务的执行，否则，将 LaunchTask（new SerializableBuffer（serializedTask)）消息发送到 CoarseGrainedExecutorBackend。

DrvierEndpoint 的 akkaFrameSize 定义和 launchTasks 方法的部分关键代码如下。

```
private val akkaFrameSize = AkkaUtils.maxFrameSizeBytes(conf)

  private def launchTasks(tasks:Seq[Seq[TaskDescription]]) {
    for (task <- tasks.flatten) {
      val serializedTask = ser.serialize(task)
      //判断如果 serialiedTask 大于等于 Akka 帧减去 Akka 预留空间大小,则调用 TaskSetManager 的 abort 方法终止该任务的执行
```

```
            if (serializedTask.limit >= akkaFrameSize - AkkaUtils.reservedSizeBytes) {
              scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>
                try {
...
                  taskSetMgr.abort(msg)
                } catch {
...
            }
            else {
              valExecutorData = ExecutorDataMap(task.ExecutorId)
              ExecutorData.freeCores -= scheduler.CPUS_PER_TASK
              //将 LaunchTask(new SerializableBuffer(serializedTask))消息发送到 CoarseGrainedExecutorBackend
              ExecutorData.ExecutorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))
            }
          }
        }
}
}
```

Akka 通过 SparkConfig 中配置的 spark.akka.frameSize 取出 Akka 帧大小的配置，默认为 128 MB，Akka 预留空间大小为 200 KB。

Akka 的 maxFrameSizeBytes 方法和预留空间代码如下。

```
def maxFrameSizeBytes(conf:SparkConf):Int = {
    //Akka 默认帧大小为 128MB
    val frameSizeInMB = conf.getInt("spark.akka.frameSize",128)
    if (frameSizeInMB > AKKA_MAX_FRAME_SIZE_IN_MB) {
      throw new IllegalArgumentException(
        s"spark.akka.frameSize should not be greater than $AKKA_MAX_FRAME_SIZE_IN_MB MB")
    }
    frameSizeInMB * 1024 * 1024
    }

  //预留空间大小为 200 KB
  val reservedSizeBytes = 200 * 1024
```

此后，CoarseGrainedExecutorBackend 匹配到 LaunchTask（data）消息后，首先调用 deserialized 方法，反序列化出 task，然后调用 Executor 的 lauchTask( )方法执行 Task 的处理。具体详情请参阅第 5 章执行器（Executor）。

Section 4.7 小结

本章内容围绕 Spark 调度器（Scheduler）的运行机制，介绍了其中涉及的重要概念，如 Spark Driver Program、Spark Job、高层调度器（DAGScheduler）、底层调度器（TaskScheduler）和调度器的通信终端（SchedulerBackend）。同时，从外围的运行框架到内部的调度器和通信终端，分别深度剖析了其各自的运行原理。并且，结合 Spark 源代码进行解析，从而加深对整个 Spark 调度器运行机制的理解。

注意，SparkContext、DAGScheduler、TaskScheduler 和 SchedulerBackend 在应用程序启动时只实例化一次，应用程序存在期间始终存在这些对象。

第 5 章　执行器（Executor）

Executor 是 Spark 中执行任务的进程，它能被不同的调度模式所调度，例如 Standalone、Mesos、Yarn。本章主要讲解 Standalone 调度模式下的 Executor，在 Standalone 调度模式下的 Executor 中，有一个 RPC（Remote Procedure Call）远程过程调用接口的引用，Spark 1.6.x 版本中，该 RPC 接口有两种不同的实现，分别是 Akka 和 Netty，目前默认使用 Netty 框架实现这个 PRC 接口，当然也可以通过 spark.rpc 配置项指定 RPC 接口的实现框架，Executor 正是通过该 RPC 接口和 Driver 通信的。为了让大家从整体上了解 Spark 中 Executor 的启动过程，给出如图 5-1 所示的时序图。

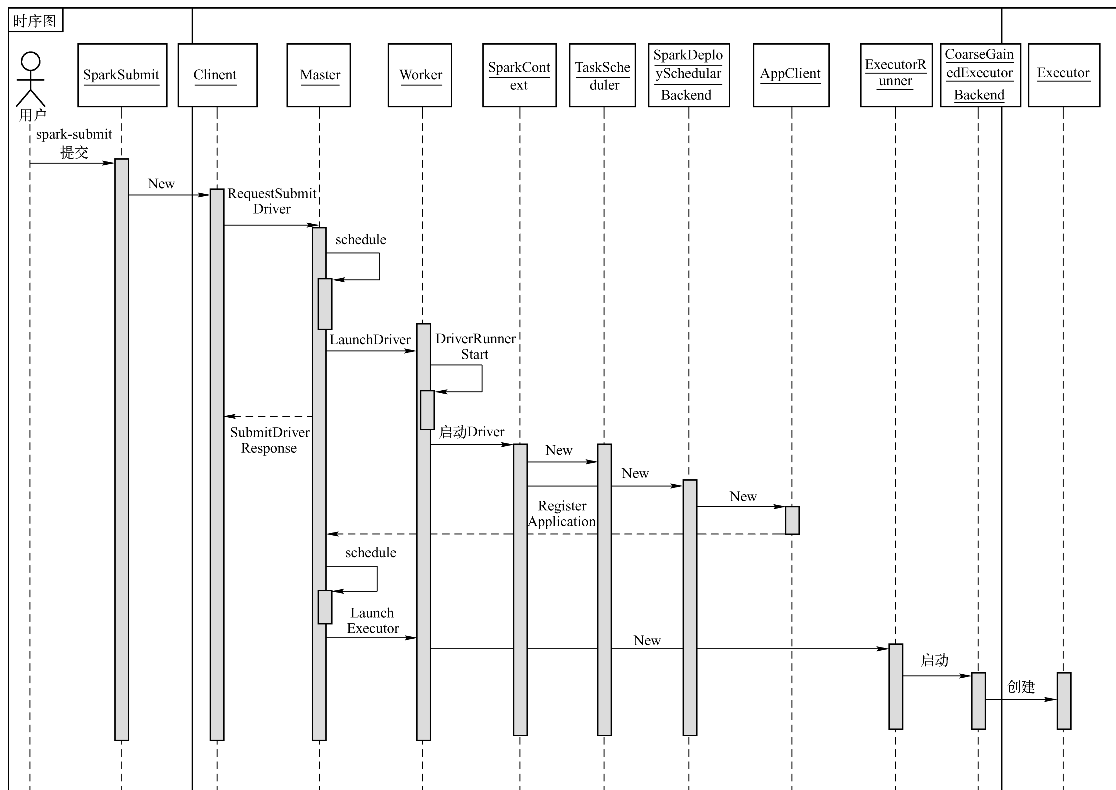


图 5-1　Executor 启动时序图

使用 spark-submit 提交程序到集群，Master 会接收到 RequestSubmitDriver 请求，之后将会调用 schedule 方法把 Driver 程序发送到 Worker 结点上运行。当 Driver 程序在一个线程中运行起来后，SparkContext 得到初始化，之后会创建 TaskScheduler 和 TaskSchedulerBackend（Standalone 模式下）。在启动 TaskSchedulerBackend 时创建 AppClient，AppClient 是一个 Ac-

tor，在其 peStart 方法中向 Master 注册。Master 收到 RegisterApplication 注册请求后，完成 Application 的注册，同时调用 schedule 方法，向 Worker 结点发送 LaunchExecutor 请求，满足请求的 Worker 结点启动线程运行 ExecutorRunner，在 ExecutorRunner 中启动 CoarseGrainedExecutorBackend，在 CoarseGrainedExecutorBackend 中创建 Executor，并完成向 Driver 的注册。

Executor 中存在一个线程池，该线程池是缓存线程池（CachedThreadPool），其特点是如果线程池长度超过处理需要，池中有空闲线程，可灵活回收空闲线程；若不能满足任务处理，则新建线程。任务被分发到 Executor 并以 TaskRunner 的形式运行于线程池中的线程之上。

Section 5.1 Executor 的创建、分配、启动及异常处理

5.1.1 Executor 的创建

上面通过时序图描述了 Executor 的调度流程，下面将深入源代码进一步分析。在 SparkContext 启动之后，SparkDeploySchedulerBackend 中会 new 出一个 AppClient，AppClient 中有一个内部类 ClientEndPoint，ClientEndPoint 继承自 ThreadSafeRpcEndpoint，其通过 RPC 机制完成和 Master 的通信。在 ClientEndPoint 的 start 方法中，会通过 registerWithMaster 方法向 Master 发送 RegisterApplication 请求，Master 收到该请求消息之后，首先通过 registerApplication 方法完成信息登记，之后将会调用 schedule 方法，在 Worker 上启动 Executor，Master 对 RegisterApplication 请求处理的源代码如下所示。

```
case RegisterApplication(description,driver) => {
      // TODO Prevent repeated registrations from some driver
      //Master 处于 STANDBY(备用)状态,不做处理
      if (state == RecoveryState.STANDBY) {
        // ignore,don't send response
      } else {
        logInfo("Registering app " + description.name)
        //由 description 描述,构建 ApplicationInfo
          val app = createApplication(description,driver)
          registerApplication(app)
          logInfo("Registered app " + description.name + " with ID " + app.id)
          //在持久化引擎中加入 application
          persistenceEngine.addApplication(app)
          //向 worker 返回注册成功的 application Id 号和 master 的 url
          driver.send(RegisteredApplication(app.id,self))
          //调用 schedule 方法,在 worker 结点上启动 Executor
          schedule()        }
    }
```

在上面的代码中，Master 匹配到 RegisterApplication 请求，先判断 Master 的状态是否为 STANDBY（备用）状态，如果不是说明 Master 为 ALIVE 状态，在这种状态下，调用 createApplication（description，sender）方法创建 ApplicationInfo，完成之后调用 persistenceEngine. addApplication（app）方法，将新创建的 ApplicationInfo 持久化，以便错误恢复。完成这两步操作之后，通过 sender！RegisteredApplication（app. id，masterUrl）向 AppClient 返回注册成功后 ApplicationInfo 的 Id 和 master 的 url 地址。

ApplicationInfo 对象是对 application 的描述，先来看一下 createApplication 这个方法的源代码，如下所示。

```
private def createApplication(desc:ApplicationDescription,driver:RpcEndpointRef):
        ApplicationInfo = {
//ApplicationInfo 创建时间
  val now = System. currentTimeMillis()
  val date = new Date(now)
  //由 date 生成 application id
  val appId = new ApplicationId(date)
//创建 ApplicationInfo
  new ApplicationInfo(now,appId,desc,date,driver,defaultCores)
 }
```

上面代码中，createApplication 方法接收 ApplicationDescription 和 ActorRef 两种类型的参数。使用 newAppicationId 方法生成 appId，关键代码如下。

```
1. val appId = "app - %s - %04d". format(createDateFormat. format(submitDate),nextAppNumber)
```

因此 appId 的格式形如：app－20160429101010－0001。在 desc 这个对象中，包含一些基本的配置，包括从系统中传入的一些配置信息，如 appname、maxCores 和 memoryPerExecutorMB 等。最后使用 desc、date、driver 和 defaultCores 等作为参数构造一个 ApplicatinInfo 对象并返回。函数返回之后，调用 registerApplication 方法，完成 Application 的注册，该方法的代码如下。

```
private def registerApplication(app:ApplicationInfo):Unit = {
    //Driver 的地址,用于 Master 和 Driver 通信
    val appAddress = app. driver. address
  //如果 addressToApp 中已经有了该 Driver 地址
  //说明该 Driver 已经注册过了,直接 return
    if (addressToApp. contains(appAddress)) {
      logInfo("Attempted to re - register application at same address:" + appAddress)
      return
    }
 //向度量系统注册
   applicationMetricsSystem. registerSource(app. appSource)
```

```
 //apps 是一个 HashSet,保存数据不能重复向 HashSet 中加入 app
   apps  +=  app
 //idToApp 是一个 HashMap,向该 HashMap 中加入 app 的 id 和 app 的对应关系
   idToApp(app.id) = app
 //endpointToApp 是一个 HashMap,记录 app 的 driver 和 app 的对应关系
   endpointToApp(app.driver) = app
 //addressToApp 是一个 HashMap,记录 app Driver 的地址和 app 的对应关系
   addressToApp(appAddress) = app
 //waitingApps 是一个数组,记录等待调度的 app 记录
   waitingApps  +=  app
 }
```

从上面的代码中可以看到，首先通过 app. driver. path. address 得到 driver 的地址，然后查看 appAdress 映射表中是否已经存在这个路径。如果存在表示该 Application 已经注册，直接返回；如果不存在，则在 waitingApps 数组中加入该 Application，同时在 idToApp、endpointToApp 和 addressToApp 映射表中加入映射关系。加入 waitingApps 数组中的 Application 等待 schedule 方法的调度。

schedule 方法有两个作用，第一，完成 Driver 的调度，将 waitingDrivers 数组中的 Driver 发送到满足运行条件的 Worker 上运行；第二，在满足条件的 Worker 结点上为 Application 启动 Executor。schedule 方法的源代码如下所示。

```
private def schedule():Unit = {
//若 Master 的状态不为 ALIVE 状态,直接返回
    if (state != RecoveryState.ALIVE) { return }
    // Drivers take strict precedence over executors
  //随机打乱 Workers 这个 HashSet 中 Worker 的次序,用于 Driver 在集群中随机加载,起到平
衡加载的作用,从这里也可以看出,Driver 是在集群中随机的 Worker 上启动的
    val shuffledWorkers = Random.shuffle(workers) // Randomization helps balance drivers
  //遍历状态为 ALIVE 的 Worker
    for (worker  <- shuffledWorkers if worker.state  ==  WorkerState.ALIVE) {
  //从 waitingDrivers 中取出 Driver,进行匹配
       for (driver  <- waitingDrivers) {
   //如果 Driver 和 Worker 匹配,内存和运行核都将得到满足
       if (worker.memoryFree  >=  driver.desc.mem && worker.coresFree  >=  driver.
desc.cores) {
 //在该 Worker 上启动加载 Driver
 launchDriver(worker,driver)
 //加载 Driver 后,从 waitingDrivers 中移除该 Driver
           waitingDrivers  -=  driver
         }
       }
     }
```




```
        startExecutorsOnWorkers()//为 application 在 worker 上启动 Executors
    }
```

在 Master 中，schedule 方法是一个很重要的方法，每一次新的 Driver 的注册、Application 的注册或者可用资源发生变动，都将调用 schedule 方法。schedule 方法用于为当前等待调度的 Application 调度可用的资源，在满足条件的 Worker 结点上启动 Executor。这个方法还有另外一个作用，就是当有 Driver 提交时，负责将 Driver 发送到一个可用资源满足 Driver 需求的 Worker 结点上运行，launchDriver（worker，driver）方法负责完成这一任务。

Application 调度成功之后，Master 将会为 Appication 在 Worker 结点上启动 Executors，调用 startExecutorsOnWorkers 方法完成此操作，其源代码如下所示。

```
private def startExecutorsOnWorkers():Unit = {
      // Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app
      // in the queue,then the second app,etc.
     //遍历没有分配满核的 Application
for (app <- waitingApps if app.coresLeft > 0) {
      //Application 中指定的每个 Executor 上占用核的个数
        val coresPerExecutor:Option[Int] = app.desc.coresPerExecutor
        //从 Workers 这个 HashSet 中过滤出资源可用的 Worker,最后调用 reverse 方法反转该
Set 集合,使 coresFree 大的 Worker 位于集合的前面,优先分配
        val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
            .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&
              worker.coresFree >= coresPerExecutor.getOrElse(1))
            .sortBy(_.coresFree).reverse
                //在 usableWorkers 中分配处理器
         val assignedCores = scheduleExecutorsOnWorkers(app,usableWorkers,spreadOutApps)
         // Now that we've decided how many cores to allocate on each worker,let's allocate them
         for (pos <- 0 untilusableWorkers.length if assignedCores(pos) > 0) {
       //为 Worker 上的 Executors 实际分配资源
            allocateWorkerResourceToExecutors(
              app,assignedCores(pos),coresPerExecutor,usableWorkers(pos))
         }
      }
}
```

在 scheduleExecutorsOnWorkers 方法中，有两种选择启动 Executor 的策略，第一种是轮流均摊策略（round-robin），采用圆桌算法依次轮流均摊，直到满足资源需求，轮流均摊策略通常会有更好的数据本地性，因此它是默认的选择策略。第二种是依次全占，在 usableWorkers 中，依次获取每个 Worker 上的全部资源，直到满足资源需求。

当使用 scheduleExecutorsOnWorkers 为 Application 分配好资源后，allocateWorkerResourceToExecutors（app，assignedCores（pos），coresPerExecutor，usableWorkers（pos））方法被调用，将会在 Worker 结点上实际分配资源。下面是 allocateWorkerResourceToExecutors 的源

代码。



```
private def allocateWorkerResourceToExecutors(
        app:ApplicationInfo,
        assignedCores:Int,
        coresPerExecutor:Option[Int],
        worker:WorkerInfo):Unit = {
    //如果 Executor 指定了核的个数,使用分配的核的个数除以指定核的个数,确定出要启动的 Executor 的个数
        val numExecutors = coresPerExecutor.map { assignedCores / _ }.getOrElse(1)
        //如果 coresPerExecutor 为 0,则取 assignedCores 并赋值给 coresToAssign
        val coresToAssign = coresPerExecutor.getOrElse(assignedCores)
    //遍历 numExecutors 中所有的 Executor
     for (i <- 1 to numExecutors) {
    //在 app 中加入 Executor 信息,这些信息包括在哪个 Worker 上分配多少个核
        val exec = app.addExecutor(worker,coresToAssign)
          //在 Worker 上启动 Executor
          launchExecutor(worker,exec)
          //启动之后将 app 状态置为 RUNNING
          app.state  = ApplicationState.RUNNING
     }
  }
```

从上面的代码中可以看到，在 for 循环中，调用了 launchExecutor（worker,exec）方法，这个方法有两个参数，第一个参数是满足条件的 WorkerInfo 信息，第二个参数是描述 Executor 的 ExecutorDesc 对象。这个方法将会向 Worker 结点发送 LaunchExecutor 的请求。launchExecutor 代码清单如下所示。

```
  private def launchExecutor(worker:WorkerInfo,exec:ExecutorDesc):Unit = {
       logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
       //向 WorkerInfo 中加入 exec 这个描述 Executor 的 eExecutorDesc 对象
       worker.addExecutor(exec)
       //向 worker 发送 LaunchExecutor 消息,加载 Executor
       //消息中携带了 masterUrl 地址、application id、Executor id、Executor 描述 desc、Executor 核的个数和 Executor 分配的内存大小
       worker.endpoint.send(LaunchExecutor(masterUrl,
         exec.application.id,exec.id,exec.application.desc,exec.cores,exec.memory))
     //向 Driver 发回 ExecutorAdded 消息,消息携带 worker 的 id 号,worker 的 host 和 port,以及分配的核的个数和内存大小
        exec.application.driver.send(
          ExecutorAdded(exec.id,worker.id,worker.hostPort,exec.cores,exec.memory))
  }
```

上面代码中，launchExecutor 有两个参数，第一个参数是 worker：WorkerInfo，代表着 Worker 的基本信息，第二个参数是 exec：ExecutorDesc，这个参数保存了 Executor 的基本配置信息，如 memory、cores 等。此方法中，有 worker. endpoint. send（LaunchExecutor(...)），这句代码的意图很明显，即向 Worker 发送 LaunchExecutor 请求，Worker 收到该请求之后将会调用方法启动 Executor。

向 Worker 发送 LaunchExecutor 消息的同时，通过 exec. application. driver！ExecutorAdded 向 Driver 发送 ExecutorAdded 消息，向 Driver 回馈 Master 都在哪些 Worker 上启动了 Executor，Executor 的编号是多少，为每个 Executor 分配了多少个核、多大的内存，以及 Worker 的联系 hostport 等消息。

Worker 收到 LaunchExecutor 消息会做哪些事情呢？从源代码中就能找到答案，在 Worker 结点中，LaunchExecutor 处理逻辑的源代码如下所示。

```
case LaunchExecutor(masterUrl,appId,execId,appDesc,cores_,memory_) =>
        //若 masterUrl 和 activeMasterUrl 不是同一个 url,说明非法的 Master 尝试加载 Executor,打印错误信息
      if (masterUrl != activeMasterUrl) {
        logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")
      } else {
        try {
          //在 workDir/appId/目录下创建以 execId 为名的 Executor 工作目录
          val executorDir = new File(workDir,appId + "/" + execId)
          //调用 mkdirs 创建目录
          if (!executorDir. mkdirs()) {
            throw newIOException("Failed to create directory " + executorDir)
          }
        //为 Executor 创建本地目录,该目录通过变量 SPARK_EXECUTOR_DIRS 设置并传递,该目录在 Application 运行结束时由 Worker 负责删除
          val appLocalDirs = appDirectories. get(appId). getOrElse {
            Utils. getOrCreateLocalRootDirs(conf). map { dir =>
              //名字前缀为 executor
              val appDir = Utils. createDirectory(dir,namePrefix = "executor")
              //更改文件目录权限为 700
              Utils. chmod700(appDir)
              //获取绝对路径
              appDir. getAbsolutePath()
            }. toSeq
          }
          //在哈希表 appDirectories 中加入 appId 和 appLocalDirs 的对应关系
          appDirectories(appId) = appLocalDirs
          //创建 ExecutorRunner
          val manager = new ExecutorRunner(
          appId,execId,
```

```
                    appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command, conf)), cores_, memory_, self, workerId, host, webUi.boundPort, publicAddress, sparkHome, executorDir, workerUri, conf, appLocalDirs, ExecutorState.RUNNING)
                //在哈希表 executors 中加入 appId + "/" + execId 和 ExecutorRunner 的对应关系
                 executors(appId + "/" + execId) = manager
                //启动 ExecutorRunner
                 manager.start()
                //Worker 上已经使用的核增加 cores_个,cores_为分配给 Executor 的核的个数
                coresUsed += cores_
                //Worker 上已经使用的内存增加 memeory_,memeory 为分配给 Executor 的内存
                memoryUsed += memory_
                 //向 Master 发送 ExecutorStateChanged 消息,该消息携带 appId,exeId,ExecutorRunner 的状态
                sendToMaster(ExecutorStateChanged(appId,execId,manager.state,None,None))
              }
}
```

从上面的代码中可以看到，首先判断传过来的 masterUrl 是否与 activeMasterUrl 相同，如果不相同，说明收到的不是处于 ALIVE 状态的 Master 发送过来的请求，这种情况下直接打印警告信息；如果相同，则说明该请求来自 ALIVE Master，于是为 Executor 创建工作目录，创建好工作目录之后，使用 appId、execId 和 appDes 等参数创建 ExecutorRunner。顾名思义，ExecutorRunner 是 Executor 运行的地方，在 ExecutorRunner 中，有一个工作线程，这个线程负责下载依赖的文件，并启动 CoarseGaindExecutorBackend。下面是 ExecutorRunner 中的线程启动的源代码。

```
private[worker] def start() {//ExecutorRunner 的 start 方法
    //创建线程
    workerThread = new Thread("ExecutorRunner for " + fullId) {
    //在线程 run 方法中调用 fetchAndRunExcutor
      override def run() {fetchAndRunExecutor() }
    }
    //启动线程
    workerThread.start()
    //终止回调函数,用于杀死进程
      shutdownHook = ShutdownHookManager.addShutdownHook { () =>
        if (state == ExecutorState.RUNNING) {
          state = ExecutorState.FAILED
        }
        killProcess(Some("Worker shutting down")) }
    }
```

在上面的代码中，定义了一个 Thread，这个 Thread 的 run 方法中调用 fetchAndRunExec-

utor 方法，负责以进程的方式启动 ApplicationDescription 中携带的 org. apache. spark. executor. CoarseGrainedExecutorBackend。fetchAndRunExecutor 方法的源代码如下所示。

```
private def fetchAndRunExecutor( ) {
    try {
      //使用 CommandUtils 创建 ProcessBuilder
      Val builder = CommandUtils. buildProcessBuilder( appDesc. command, new SecurityManager( conf), memory, sparkHome. getAbsolutePath, substituteVariables)
    //得到进程启动命令
    val command = builder. command( )
    //格式化启动命令
      val formattedCommand = command. asScala. mkString( "\"", "\" \"", "\"")
      logInfo( s"Launch command: $ formattedCommand")
      //设置进程工作目录
       builder. directory( executorDir)
      //在进程环境变量中加入 SPARK_EXECUTOR_DIRS 工作目录
builder. environment. put( "SPARK_EXECUTOR_DIRS", appLocalDirs. mkString( File. pathSeparator) )
     //在进程环境变量中加入 SPARK_LAUNCH_WITH_SCALA
       builder. environment. put( "SPARK_LAUNCH_WITH_SCALA", "0")
     //Web 日志的 url
       val baseUrl =
         s"http:// $ publicAddress: $ webUiPort/logPage/? appId = $ appId&executorId = $ execId&logType = "
       builder. environment. put( "SPARK_LOG_URL_STDERR", s" $ {baseUrl} stderr")
       builder. environment. put( "SPARK_LOG_URL_STDOUT", s" $ {baseUrl} stdout")
     //启动 CoarseGrainedExecutorBackend 进程
       process = builder. start( )
       val header = "Spark Executor Command:%s\n%s\n\n". format(
       formattedCommand, " = " * 40)
       val stdout = new File( executorDir, "stdout")
       stdoutAppender = FileAppender( process. getInputStream, stdout, conf)
       val stderr = new File( executorDir, "stderr")
       Files. write( header, stderr, UTF_8)
       stderrAppender = FileAppender( process. getErrorStream, stderr, conf)
      //等待进程退出
       val exitCode = process. waitFor( )
       state  = ExecutorState. EXITED
       val message = "Command exited with code " + exitCode
      //Executor 进程退出后向 Worker 发送 ExecutorStateChanged 消息,Worker 收到该消息后回收分配给该 Executor 的 cores 和 memory
     worker. send( ExecutorStateChanged( appId, execId, state, Some( message), Some( exitCode) ) )
```

```
36.        }
37. }
```

CoarseGrainedExecutorBackend 启动后，会首先通过传入的 driverUrl 这个参数向 DriverActor 发送 RegisterExecutor（executorId, self, hostPort, cores, extractLogUrls），Driver 收到此消息后，登记注册好的 Executor 信息，并回复 RegisteredExecutor，CoarseGrainedExcutorBackend 收到 RegisteredExecutor 消息后，创建 org. apache. spark. executor. Executor，至此，Executor 创建完毕。

5. 1. 2 Executor 的资源分配

Executor 作为单独的进程运行于 Worker 结点上，通过图 5-1 可以清楚地看到，当 ExecutorRunner 启动 CoarseGrainedExecutorBackend 进程时，在 CoarseGrainedExecutorBackend 中会先向 Driver 发送 RegisterExecutor 请求，完成注册。如果注册成功，返回 RegisteredExecutor 消息，CoarseGrainedExecutor 收到该消息并新建 Executor。如果注册失败，返回 RegisterExecutorFailed 消息，CoarseGrainedExecutorBackend 进程退出。

CoarseGrainedExecutorBackend 此时以进程的形式启动，运行于一个独立的 JVM 中。编写过 Java 程序的人都知道，在启动 JVM 时，可以通过 java – server 配置项为 JVM 分配堆栈大小，因此，此处 ProcessBuilder 启动的 CoarseGrinedExecutorBackend 进程也可以通过类似的命令配置 JVM 的堆栈大小，那么这些堆栈的大小是如何设置的呢？

Executor 运行在一个单独的进程中，必然会涉及资源的分配问题，也可以像运行 Java 程序一样，在启动进程时，为其分配资源。在 Spark 中，可以通过指定配置参数的形式，为 Executor 配置运行资源，例如，可以指定每一个 Executor 占用内存的大小、运行核的个数等。表 5-1 所示是 Executor 的一些配置参数。

表 5-1　Executor 配置参数

参数名称	默认值	描述
spark. executor. memory	1g	为 Executor 分配的内存大小
spark. executor. extraJavaOptions	none	传递给 Executor 的 JVM 选项，如 GC 设置、Heap 设置等
spark. executor. extraLibraryPath	none	指定启动 Executor JVM 时依赖的库路径
spark. executor. logs. rolling. maxSize	none	设置 Executor 日志滚动的大小
spark. executorEnv. [EnvironmentVariableName]	none	为 Executor 指定环境变量
spark. executor. cores	1 或所有	Executor 占用核的个数，Yarn 模式默认占用一个，Standalone 默认使用所有核
spark. executor. heartbeatInterval	10 s	向 Driver 发送心跳的时间间隔
spark. executor. port	random	Executor 监控的端口，用于和 Driver 通信
spark. dynamicAllocation. maxExecutors	无穷	动态分配 Executor 的最大个数
spark. dynamicAllocation. minExecutors	0	动态分配 Executor 的最小个数

Executor 的这些配置项可以有多种不同的配置方式，例如，可以在系统环境变量中指定 spark. executor. memory 的大小，也可以在配置文件 spark - env. sh 中指定 spark. executor. memory 的大小，最常用的方式是在使用 spark - submit 提交程序时，通过参数指定 Executor 的配置，如果不指定 Executor 的配置，将使用默认值。

通过 spark - submit 指定 Executor 参数，可以使用如【例 5-1】所示的参数提交程序。

【例 5-1】 spark - submit 中 Executor 参数的指定。

```
./bin/spark-submit \
   --class org.apache.spark.examples.SparkPi \
   --master spark://207.184.161.138:7077 \
   --executor-memory 20G \   //指定 Executor 运行内存
   --total-executor-cores 100 \ //指定 Executor 运行内核的个数
  /path/to/examples.jar \
  1000
```

【例 5-1】中，通过 --executor - memory 指定 Executor 内存大小为 20GB，有可执行的内核的个数为 100 个。

这些配置的参数是如何传递给 CoarseGrainedExecutorBackend，并以这些传递过来的参数启动 JVM 进程呢？首先来看一下在程序中是如何接收这些命令行参数的。

查看 spark - submit 脚本，脚本中直接运行了 org. apache. spark. deploy. SparkSubmit 这个对象。spark - submit 脚本的内容如下所示。

```
#! /usr/bin/env bash
SPARK_HOME="$(cd "dirname "$0""/..;pwd)"
export PYTHONHASHSEED=0
exec "$SPARK_HOME"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"//运行 SparkSubmit
```

进入到 SparkSubmit 中，main 函数的代码如下所示。

```
def main(args:Array[String]):Unit = {
    //由启动 main 函数传入的参数构建 SparkSubmitAruments 对象
     val appArgs = new SparkSubmitArguments(args)
    //打印参数信息
     if (appArgs.verbose) {
       printStream.println(appArgs)
     }
     appArgs.action match {
       //提交,调用 submit 方法
       case SparkSubmitAction.SUBMIT  => submit(appArgs)
       //杀死,调用 kill 方法
       case SparkSubmitAction.KILL  => kill(appArgs)
```

```
        //请求状态,调用 requestStatus 方法
        case SparkSubmitAction. REQUEST_STATUS  =>  requestStatus(appArgs)
        }
    }
```

上面代码中，spark－submit 脚本提交的命令行参数通过 main 函数的 args 获取，并将 args 参数传入 SparkSubmitArguments 中解析完成。最后通过匹配 appArgs 参数中的 action 类型执行 submit、kill 和 requestStatus 操作。

进入到 SparkSubmitArguments 中，分析一下参数的解析过程。SparkSubmitArguments 中的关键代码如下所示。

```
//调用 parse 方法,从命令行中解析出各个参数
   try {
     parse(args. asJava)
   } catch {
     //捕获到 IllegalArgumentException,打印错误并退出
     case e:IllegalArgumentException  =>
       SparkSubmit. printErrorAndExit(e. getMessage( ))
   }
   //合并默认的 Spark 配置项,使用传入的配置覆盖默认的配置
  mergeDefaultSparkProperties( )
  //从 sparkProperties 移除不是以"spark. "为开始的配置
  ignoreNonSparkProperties( )
  //加载系统环境变量中的配置信息
  loadEnvironmentArguments( )
  //验证参数是否合法
  val idateArguments( )
```

在上面的代码中，parse（args. toList）将会解析命令行参数，通过 mergeDefaultSpark-Properties 合并默认配置，调用 ignoreNonSparkProperties 方法忽略不是以“spark.”为开始的配置，方法 loadEnvironmentArguments 加载系统环境变量，最后调用 validateArguments 方法检验参数的合法性。这些配置如何提交呢？main 函数中有 case SparkSubmitAction. SUBMIT => submit（appArgs），这句代码判断是否提交参数并执行程序，如果匹配到 SparkSubmitAction. SUBMIT，则调用 submit（appArgs）方法，参数 appArgs 是 SparkSubmitArguments 类型，appArgs 中包含了提交的各种参数，包括命令行传入及默认的配置项。

submit（appArgs）方法主要完成以下两件事情。

1）准备提交环境，

2）执行 main 方法完成提交。

首先看一下在 Spark 中是如何准备环境的。在 submit（appArgs）方法中，有以下源代码。

```
val (childArgs,childClasspath,sysProps,childMainClass) = prepareSubmitEnvironment(args)
```

在这段代码中，调用 prepareSubmitEnvironment（args）方法完成提交环境的准备。该方法返回一个四元 Tuple，分别表示子进程参数、子进程 classpath 列表、系统属性 map 和子进程 main 方法。完成了提交环境的准备工作之后，接下来将启动子进程，在 Standalone 模式下，启动的子进程是 org. apache. spark. deploy. Client 对象。具体执行过程在 runMain 函数中，关键代码如下所示。

```
private def runMain(
          childArgs:Seq[String],
          childClasspath:Seq[String],
          sysProps:Map[String,String],
          childMainClass:String,
          verbose:Boolean):Unit = {
  …
   Thread. currentThread. setContextClassLoader(loader)//获得 classLoader
     for (jar  <- childClasspath) {//遍历 classpath 列表
         addJarToClasspath(jar,loader)//使用 loader 类加载器将 jar 包依赖加入 classpath
  }
  for ((key,value)  <- sysProps) {//将 sysProps 中的配置全部设置到 System 全局变量中
       System. setProperty(key,value)
  }
  var mainClass:Class[_] = null
  mainClass = Utils. classForName(childMainClass)//获取启动的 MainClass
…//得到启动的对象的 main 方法
  val mainMethod = mainClass. getMethod("main",new Array[String](0). getClass)
…//使用反射执行 main 方法,并将 childArgs 作为参数传入该 main 方法
  mainMethod. invoke(null,childArgs. toArray)
  }
```

在上面的代码中，使用 Utils 工具提供的 classForName 方法找到主类，然后在 mainClass 上调用 getMethod 方法得到 main 方法，最后在 mainMethod 上调用 invoke 执行 main 方法。需要注意的是，执行 invoke 方法的同时传入了 childArgs 参数，这个参数中保留了配置信息。Utils. classForName（childMainClass）方法将返回要执行的主类，这里的 childMainClass 是哪一个类呢？其实这个参数在不同的部署模式下是不一样的，standalone 模式下，childMainClass 指的是 org. apache. spark. deploy. Client 这个类，从源代码中可以找到依据，源代码如下所示。

```
//在 prepareSubmitEnvironment 方法中判断是否为 Standalone 集群模式
if (args. isStandaloneCluster) {
      //判断使用 Rest childMainClass 为 org. apache. spark. deploy. rest. RestSubmissionClient
      if (args. useRest) {
         childMainClass = "org. apache. spark. deploy. rest. RestSubmissionClient"
         childArgs  += (args. primaryResource,args. mainClass)
```

```
      } else {
        //非 Rest,childMainClass 为 org. apache. spark. deploy. Client
        childMainClass = "org. apache. spark. deploy. Client"
        if (args. supervise) {childArgs  +=  " --supervise" }
        //设置 driver memory
        Option(args. driverMemory). foreach { m  =>  childArgs  +=  (" --memory",m) }
        //设置 driver cores
        Option(args. driverCores). foreach { c  =>  childArgs  +=  (" --cores",c) }
        childArgs  +=  "launch"
        childArgs  +=  (args. master,args. primaryResource,args. mainClass)
      }
      if (args. childArgs ! =  null) {
        childArgs  + +=  args. childArgs
      }
    }
```

在上面的代码中，程序首先根据 args. isStandaloneCluster 判断部署模式，如果是 Standalone 模式并且不使用 REST 服务，childMainClass = "org. apache. spark. deploy. Client"。上述代码中，可以看出 childArgs 中存入了 Executor 的 memory 配置和 cores 配置。如 runMain 方法中描述的一样，程序将启动 org. apache. spark. deploy. Client 这个类，并运行主方法，Client 类中做了哪些事情，先来看一下这个类中是如何完成调用的。下面是 Client 对象及主方法。

```
object Client {
  def main(args:Array[String]) {
    //若 sys 中不包含 SPARK_SUBMIT,打印警告信息
    if (! sys. props. contains("SPARK_SUBMIT")) {
      println("WARNING:This client is deprecated and will be removed in a future version of
Spark")
    //打印提示信息
    println("Use ./bin/spark -submit with \" --master spark://host:port\"")
  }
  //创建 SparkConf 对象
  val conf = new SparkConf()
   //创建 ClientArguments 对象,代表 Driver 端的参数
    val driverArgs = new ClientArguments(args)
    if (!driverArgs. logLevel. isGreaterOrEqual(Level. WARN)) {
      conf. set("spark. akka. logLifecycleEvents","true")
    }
    //设置 RPC 请求超时时间为 10 s
    conf. set("spark. rpc. askTimeout","10")
    //设置 Akka 的日志级别,使用 WARNING 代替 WARN
    conf. set("akka. loglevel",driverArgs. logLevel. toString. replace("WARN","WARNING"))
```

```
20.         Logger.getRootLogger.setLevel(driverArgs.logLevel)
21.         //使用 RpcEnv 的 create 创建 Rpc 环境
22.         val rpcEnv =
23.     RpcEnv.create("driverClient",Utils.localHostName(),0,conf,new SecurityManager(conf))
24.     //得到 master 的 url 并得到 Master 的 Endpoints,用于与 Master 通信
25.     val masterEndpoints = driverArgs.masters.map(RpcAddress.fromSparkURL).
26.     map(rpcEnv.setupEndpointRef(Master.SYSTEM_NAME,_,Master.ENDPOINT_NAME))
27.     //使用 RpcEvn 的 setupEndpoint 方法设置名为 client 的 Endpoint,该 Endpoint 可以与 Master
通信
28.     rpcEnv.setupEndpoint("client",new ClientEndpoint(rpcEnv,driverArgs,masterEndpoints,conf))
29.     //等待 rpcEnv 的终止
30.     rpcEnv.awaitTermination()
31.       }
32.     }
```

上面代码中，首先实例化出一个 SparkConfig 对象，通过这个配置对象，可以在代码中指定一些配置项，如 appName、Master 地址等。val driverArgs = new ClientArguments（args）使用传入的 args 参数构建一个 ClientArguments 对象，该对象同样保留传入的配置信息，Executor memory、Executor cores 等都包含在这个对象中。

上面代码的第 22 行，使用 RpcEnv.create 工厂方法创建一个 rpcEnv 成员，使用该成员设置好到 Master 的通信端点，通过该端点实现与 Master 的通信。Spark 之前版本采用 Akka 框架来实现 Rpc 远程过程调用，通过使用 Akka 的分布式异步通信机制，完成各结点之 间的通信。这里通过 masterEndpoint 向 Master 发送 RequestSubmitDriver（driverDescription）请求，完成 Driver 的注册。

代码中，提交的配置参数始终在不同的对象和结点上传递。Master 把 Driver 加载到 Worker 结点并启动起来，Worker 结点上运行的 Driver 同样包含配置参数。当 Driver 端的 SparkContext 启动并实例化 DAGScheduler、TaskScheduler 时，TaskSchedulerBackend 在做另一件事情，即实例化 AppClient，AppClient 中有 AppClientPoint，用于通信。AppClientPoint 的 onStart 方法中，向 Master 发送 RegisterApplication（appDescription,self）请求，Master 结点收到请求并调用 schedule 方法，向 Worker 发送 LaunchExecutor（masterUrl，exec.application.id，exec.id，exec.application.desc,exec.cores,exec.memory）请求，Worker 结点启动 ExecutorRunner，ExecutorRunner 启动 CoarseGrainedExecutorBackend 并向 Driver 注册。

在 CoarseGrainedExecutorBackend 的 main 方法中，有如下所示的代码。

```
var argv = args.toList//将 args 转化成 List
      while (!argv.isEmpty) {//argv 不为空,则一直循环
        argv match {
          case ("--driver-url") :: value :: tail =>
            driverUrl = value//得到 Driverurl
            argv = tail
          case ("--executor-id") :: value :: tail =>
```

```
            executorId = value//得到 executor id
            argv = tail
          case ("--hostname") ::value ::tail =>
           hostname = value//得到 hostname
           argv = tail
           case ("--cores") ::value ::tail =>
             cores = value.toInt//得到配置的 Executor 核的个数
             Sargv = tail
           case ("--app-id") ::value ::tail =>
             appId = value//得到 application 的 id
             argv = tail
           case ("--worker-url") ::value ::tail =>
             workerUrl = Some(value)//得到 worker 的 url
             argv = tail
           case ("--user-class-path") ::value ::tail =>
             userClassPath += new URL(value)//得到用户类路径
             argv = tail
           case Nil =>
           case tail =>
             System.err.println(s"Unrecognized options: ${tail.mkString(" ")}")
             printUsageAndExit()//打印并退出
        }
    }
```

从程序提交一直到 CoarseGrainedExecutorBackend 进程启动，配置参数一直被传递。在 CoarseGrainedExecutorBackend 中取出了内核配置信息，并通过 run（driverUrl，executorId，hostname，cores，appId，workerUrl，userClassPath）将该配置信息传入 run 方法，此时 CoarseGrainedExecutorBackend 以进程的形式在 JVM 中启动，JVM 以占用指定核的个数启动起来。需要注意的是，在一个 Worker 结点上，只要处理内核的个数满足 Executor 启动要求，一个 Worker 结点上可以运行多个 Executor。

5.1.3 Executor 的启动

在 5.1.2 节中，了解了参数的传递过程和外部配置的参数，通过结点之间的消息传递，最终作用在 Executor 上，外部配置的参数终将会在 Executor 上发挥作用。在本节中，主要探讨 Executor 的启动过程。

再回到 SparkDeploySchedulerBackend 上来，在 SparkDeploySchedulerBackend 中会新建 AppClient，AppClient 启动后会向 Master 发送 RegisterApplication 消息，Master 在收到 RegisterApplication 请求后，将会调用 schedule 方法，在调度过程中会使用 allocateWorkerResourceToExecutors 方法为 Worker 结点上的 Executor 分配资源。分好资源后，调用 launchExecutor（worker,exec）方法，此方法将向 Worker 发送 LaunchExecutor 请求，启动 Executor。下面是

launchExecutor 方法的源代码。

```
private def launchExecutor(worker:WorkerInfo,exec:ExecutorDesc):Unit = {
    logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
    //向 WorkerInfo 中加入 exec 这个描述 Executor 的 eExecutorDesc 对象
    worker.addExecutor(exec)
    //向 worker 发送 LaunchExecutor 消息,加载 Executor
    //消息中携带了 masterUrl 地址、application id、Executor id、Executor 描述 desc、Executor 核的个数和 Executor 分配的内存大小
    worker.endpoint.send(LaunchExecutor(masterUrl,
      exec.application.id,exec.id,exec.application.desc,exec.cores,exec.memory))
        //向 Driver 发回 ExecutorAdded 消息,消息携带 worker 的 id 号,Worker 的 host 和 port,以及分配的核的个数和内存大小
        exec.application.driver.send(
           ExecutorAdded(exec.id,worker.id,worker.hostPort,exec.cores,exec.memory))
}
```

launchExecutor 方法中会向 Worker 发送 LaunchExecutor 消息，先来看一看 LaunchExecutor 消息的几个参数，masterUrl 是集群中 master 的 Url，exec.application.id 是 Master 为 Application 分配的 id 号，该 id 号在注册 Application 时就已经生成好了。exec.id 是 Executor 的 id 号，Master 在为 Application 分配 Worker 时该 id 号就已经指定好。exec.application.desc 是 application 的描述，包括 name、maxCores、memoryPerExecutorMB、command、appUiUrl 和 eventLogDir 等信息。exec.cores、exec.memory 分别代表 Executor 运行核的个数和运行内存。

Worker 收到 LaunchExecutor 消息后，首先启动 ExecutorRunner，然后向 Master 发送 ExecutorStateChanged（appId,execId,manager.state,None,None）消息，Master 在收到此消息后将视 Executor 的状态调用 schedule()方法，重新调整集群资源。

ExecutorRunner 启动后，将调用 fetchAndRunExecutor 方法创建 Executor 工作目录并启动 CoarseGraindExecutorBancked 进程，fetchAndRunExecutor 方法的源代码如下所示。

```
private def fetchAndRunExecutor() {
    try {
      //使用 CommandUtils 创建 ProcessBuilder
      Val builder = CommandUtils.buildProcessBuilder(appDesc.command,new SecurityManager(conf),memory,sparkHome.getAbsolutePath,substituteVariables)
      //得到进程启动命令
      val command = builder.command()
      //格式化启动命令
      val formattedCommand = command.asScala.mkString("\"","\" \"","\"")
      logInfo(s"Launch command: $ formattedCommand")
      //设置进程工作目录
      builder.directory(executorDir)
      //在进程环境变量中加入 SPARK_EXECUTOR_DIRS 工作目录
```

```
13.            builder.environment.put("SPARK_EXECUTOR_DIRS",appLocalDirs.mkString(File.pathSeparator))
14.         //在进程环境变量中加入 SPARK_LAUNCH_WITH_SCALA
15.         builder.environment.put("SPARK_LAUNCH_WITH_SCALA","0")
16.         //Web 日志的 url
17.         valbaseUrl =
18.         s"http://$publicAddress:$webUiPort/logPage/?appId=$appId&executorId=$execId&logType="
19.         builder.environment.put("SPARK_LOG_URL_STDERR",s"${baseUrl}stderr")
20.         builder.environment.put("SPARK_LOG_URL_STDOUT",s"${baseUrl}stdout")
21.         //启动 CoarseGrainedExecutorBackend 进程
22.         process = builder.start()
23.         val header = "Spark Executor Command:%s\n%s\n\n".format(
24.         formattedCommand," = " * 40)
25.         val stdout = new File(executorDir,"stdout")
26.   stdoutAppender = FileAppender(process.getInputStream,stdout,conf)
27.         val stderr = new File(executorDir,"stderr")
28.         Files.write(header,stderr,UTF_8)
29.         stderrAppender = FileAppender(process.getErrorStream,stderr,conf)
30.         //等待进程退出
31.         val exitCode = process.waitFor()
32.         state  = ExecutorState.EXITED
33.         val message = "Command exited with code " + exitCode
34.         //Executor 进程退出后向 Worker 发送 ExecutorStateChanged 消息,Worker 收到该消息后回收分配给该 Executor 的 cores 和 memory
35.       worker.send(ExecutorStateChanged(appId,execId,state,Some(message),Some(exitCode)))
36.     }
37. }
```

在上面代码中的第 4 行，使用 CommandUtils 工具构建了一个 ProcessBuilder，构建 ProcessBuilder 时传入了 appDesc.command、memory、sparkHome.getAbsolutePath 和 substituteVariables 这几个参数，分别代表命令、内存大小、SparkHome 绝对路径和替代参数。appDesc.command 命令对象里面包括 mainClass、Arguments、Environment、classPathEntries、libraryPathEntries 和 javaOpts 这几项内容。command 中包含了启动进程的关键配置项，其中 mainClass 代表的就是启动进程的名称，Standalone 模式下 mainClass 指代的是“org.apache.spark.executor.CoarseGrainedExecutorBackend”。Command 的源代码如下所示。

```
private[spark] case class Command(
    mainClass:String,//启动进程主类名称
    arguments:Seq[String],//参数
    environment:Map[String,String],//环境变量
```

```
        classPathEntries:Seq[String],//类路径
        libraryPathEntries:Seq[String],//库路径
        javaOpts:Seq[String]){//JVM 配置项
}
```

这些参数在构建提交环境时就已经准备好，通过 ApplicationDescription 传递过来。上面提到的替代参数是什么意思呢？这还得从部署谈起。在 Standalone 模式下，TaskScheduler 启动的 SchedulerBackend 是 SparkDeploySchedulerBackend。SparkDeploySchedulerBackend 会预先构建 Executor 的一些参数，在构建这些参数时，这些参数的值还没法获取，因此使用了替代参数，在这些参数的值获取了之后再替代。在 SparkDeloySchedulerBackend 中有以下这些事先构建好的替代参数，源代码如下所示。

```
val args = Seq(
      "--driver-url",driverUrl,
      "--executor-id","{{EXECUTOR_ID}}",//Executor Id,在构建该参数时,Executor 还没有构建好
      "--hostname","{{HOSTNAME}}",//主机名称
      "--cores","{{CORES}}",//核的个数
      "--app-id","{{APP_ID}}",//application id 号
      "--worker-url","{{WORKER_URL}}")//worker 的 url
```

上面代码中，构建了一个名为 args 的序列。该序列中包括 driver-url、executor-id、hostname、cores、app-id 和 worker-url 这几个参数，并指定使用{{name}}作为占位符，暂时代替这些还没有具体值的参数。这些参数将会在 org.apache.spark.deploy.worker.ExecutorRunner 启动 CoarseGrainedExecutorBackend 时被替换成真实的值。替换源代码如下所示。

```
while (!argv.isEmpty) {
      rgv match {
        case ("--driver-url") ::value ::tail =>
          driverUrl = value
          argv = tail
        case ("--executor-id") ::value ::tail =>//替换 executor 的值
          executorId = value
          argv = tail
        case ("--hostname") ::value ::tail =>//替换 hostname
         hostname = value
          argv = tail
        case ("--cores") ::value ::tail =>//替换 cores
          cores = value.toInt
          argv = tail
        case ("--app-id") ::value ::tail =>//替换 application id
          appId = value
```

```
            argv = tail
        case ("--worker-url") :: value :: tail => //替换 worker url
            workerUrl = Some(value)
            argv = tail
        case ("--user-class-path") :: value :: tail =>
            userClassPath += new URL(value)
            argv = tail
        case Nil =>
        case tail =>
            printUsageAndExit()
    }
}
```

再回过头来看一下前面 fetchAndRunExecutor 方法中的第 22 行代码，process = builder. start()，这句代码调用 Java 的 ProcessBuilder 的 start 方法，依据传入的启动参数启动进程。在使用 CommondUtils 的 buildProcessBuilder 方法创建 builder 时，传入了启动进程的命令，该命令已经在 SparkDeploySchedulerBackend 中构建好，源代码如下所示。

```
val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
        args, sc.executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)
```

从上面代码中可见，在构建 Command 时，传给 Command 的第一个参数是 org. apache. spark. executor. CoarseGrainedExecutorBackend，该参数表示 mainClass。因此 builder. start 启动的进程就是 org. apache. spark. executor. CoarseGrainedExecutorBackend 进程，并且在启动中指定了进程所占处理内核的个数、内存及 JVM 配置信息。

在 CoarseGrainedExecutorBackend 进程启动后，将会向 Driver 端发起注册请求。之所以要向 Driver 注册，是因为实际控制 Executor 计算任务的还是 Driver，Master 只是间接地为 Driver 分配了 Executor，分配好了之后，使用权便交到 Driver 手中。Driver 要知道 Master 都为自己分配了哪些 Executor，这些 Executor 都位于哪些 Worker 中，因此 Executor 在 Worker 上启动成功后，主动去联系它的“主人”，向 Driver 注册，以免“主人”长时间等待 Master 分配的资源。RegisterExecutor 消息在 CoarseGrainedExecutorBackend 的 onStart 方法中发出，源代码如下所示。

```
override def onStart() {
    logInfo("Connecting to driver:" + driverUrl)
    rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
      // This is a very fast action so we can use "ThreadUtils.sameThread"
      driver = Some(ref)
    //向 Driver 发送 ask 请求,等待 Driver 的回应
      ref.ask[RegisterExecutorResponse](
        RegisterExecutor(executorId, self, hostPort, cores, extractLogUrls))
```

```
9.        }(ThreadUtils.sameThread).onComplete {
10.           // This is a very fast action so we can use "ThreadUtils.sameThread"
11.          //成功
12.           case Success(msg) => Utils.tryLogNonFatalError {
13.             Option(self).foreach(_.send(msg)) // msg must be RegisterExecutorResponse
14.           }
15.          //失败
16.           case Failure(e) => {
17.             logError(s"Cannot register with driver: $driverUrl", e)
18.             System.exit(1)
19.           }
20.        }(ThreadUtils.sameThread)
21.      }
```

上面代码中的第 3 行使用 rpcEnv 得到 Driver 的一个端点引用，然后使用 ask 方法向 Driver 发出 RegisterExecutor（executorId,self,hostPort,cores,extractLogUrls）请求，Driver 端收到请求后完成 Executor 的注册，并返回 RegisteredExecutor 消息，CoarseGrainedExecutorBackend 的 receive 方法收到 RegisteredExecutor 消息后，会立即创建一个 Executor 执行器。源代码如下所示。

```
case RegisteredExecutor(hostname) =>
      logInfo("Successfully registered with driver")
      //创建 Executor 执行器
      executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
```

上面代码中，匹配到 RegisterdExecutor 消息，证明 CoarseGraindedExecutorBackend 在 Driver 端已经注册，并打印出注册成功的信息，然后后创建 Executor。Executor 将协助 CoarseGraindedExecutorBackend 完成计算任务。

Executor 在启动和执行计算任务过程中都有可能遇到错误或者异常，下一节将讲解在 Executor 中是如何进行异常处理的。

5.1.4 Executor 的异常处理

Executor 在启动和执行计算任务过程中都有可能遇到错误或者异常，会遇到哪些错误呢？看一下 Executor 的 run 方法就知道了，在 run 方法中通过 try－catch 语句捕获了常见的异常，这些异常包括 FetchFailedException、TaskKilledException、CommitDeniedException、任何可 Throwable 的异常。Executor 中是如何处理异常的呢？看一下源代码吧，下面是捕获异常的代码。

```
      // FetchFailedException:抓取 shuffle Block 时发生异常
      case ffe: FetchFailedException =>
          val reason = ffe.toTaskEndReason
```

```
            execBackend.statusUpdate(taskId,TaskState.FAILED,ser.serialize(reason))
        // TaskKilledException:当一个任务被显式地结束时抛出该异常
          case _:TaskKilledException | _:InterruptedException if task.killed =>
            logInfo(s"Executor killed $taskName (TID $taskId)")
            execBackend.statusUpdate(taskId,TaskState.KILLED,ser.serialize(TaskKilled))
        //CommitDeniedException:当 task 尝试向 HDFS 写出任务但被 Driver 拒绝时抛出该异常
          case CDE:CommitDeniedException =>
            val reason = cDE.toTaskEndReason
            execBackend.statusUpdate(taskId,TaskState.FAILED,ser.serialize(reason))
          //Throwable:其他类型的异常
          case t:Throwable =>
            logError(s"Exception in $taskName (TID $taskId)",t)
            val metrics:Option[TaskMetrics] = Option(task).flatMap { task =>
              task.metrics.map { m =>
                m.setExecutorRunTime(System.currentTimeMillis() - taskStart)
                m.setJvmGCTime(computeTotalGcTime() - startGCTime)
                m.updateAccumulators()
                m
              }
            }
```

上面代码中，遇到异常分别做以下处理。

- FetchFailedException：抓取 shuffle Block 时发生异常。捕获到这个异常之后，将会传递给 DAGScheduler，并且 DAGScheduler 将重新提交前一个 Stage。捕获到该异常后，会调用 CoarseGrainedExecutorBackend 的 statusUpdate 方法，这个方法将向 Driver 发送 StatusUpdate 消息，消息中包括 taskId、executorId、taskState，以及捕获的异常堆栈信息。Driver 匹配到 StatusUpdate 消息，调用 TaskSchedulerImpl 的 statusUpdate 方法，完成更新任务相关登记信息并重新调度资源执行失败任务。
- TaskKilledException：当一个任务被显式地结束时抛出该异常，例如：任务失败，线程被中断。当捕获到该异常时，会执行 execBackend.statusUpdate（taskId,TaskState.KILLED,ser.serialize(TaskKilled)），该语句调用了 CoarseGrainedExecutorBackend 的 statusUpdate 方法，并将 taskId、TaskState.KILLED 及错误信息传入该方法，最后通过 driverRef.send(msg) 的方式将 StateUpdate（executorId,tasked,state,data）消息发送到 CoarseGrainedExecutorBackend 中，在 CoarseGrainedExecutorBackend 的 receive 方法中匹配到 StatusUpdate（executorId,taskId,state,data）消息，将会执行与 FetchFailedException 类似的操作，即更新登记信息表并重新调度资源运行任务。
- CommitDeniedException：当 task 尝试向 HDFS 写出任务但被 Driver 拒绝时抛出该异常。该异常的处理与 FetchFailedException 完全一样。
- Throwable：其他类型的异常，将终止 Executor，并重新调度资源。

Section

5.2 执行器的通信接口（ExecutorBackend）

ExecutorBackend 是 Executor 向集群发送更新消息的一个可插拔的接口。ExecutorBackend 拥有不同的实现，Standalone 模式下 ExecutorBackend 的默认实现是 CoarseGrainedExecutorBackend；Local 模式下 ExecutorBackend 的默认实现是 LocalBackend；Mesos 调度模式下 ExecutorBackend 的默认实现是 MesosExecutorBackend。本节主要探讨 Standalone 模式下的 ExecutorBackend，通过源代码深入理解 ExecutorBackend 接口设计的精髓。

5.2.1 ExecutorBackend 接口与 Executor 的关系

在此详细分析在 Standalone 模式下 ExecutorBackend 和 Executor 的关系。在 SparkDeploySchedulerBackend 中会实例化一个 AppClient。AppClient 中携带了 command 信息，在 command 信息中指定了要启动的 ExecutorBackend 的实现类。在 Standalone 模式下，该 ExecutorBackend 的实现类是 org. apache. spark. executor. CoarseGrainedExecutorBackend 类。SparkDeploySchedulerBackend 中实例化 AppClient 的源代码如下所示。

```
1.  //command 中指定 ExecutorBackend 的实现类
2.  val command = Command( "org. apache. spark. executor. CoarseGrainedExecutorBackend",
3.  args,sc. executorEnvs,classPathEntries  + +  testingClassPath,libraryPathEntries,javaOpts)
4.    val appUIAddress = sc. ui. map(_. appUIAddress). getOrElse( "" )
5.    val coresPerExecutor = conf. getOption( "spark. executor. cores" ). map(_. toInt)
6.    //使用 Commond 等信息构建 application 的描述对象 ApplicationDescription
7.  val appDesc = new ApplicationDescription( sc. appName,maxCores,sc. executorMemory,
8.      command,appUIAddress,sc. eventLogDir,sc. eventLogCodec,coresPerExecutor)
9.  //将 appDesc 作为参数,创建 AppClient
10. client = new AppClient( sc. env. rpcEnv,masters,appDesc,this,conf)
11. //调用 start 方法,启动 AppClient
12.   client. start( )
13. //将状态设置成 SUBMITTED
14.   launcherBackend. setState( SparkAppHandle. State. SUBMITTED)
15. //等待登记注册
16.   waitForRegistration( )
17. //将状态设置成 RUNNING
18.   launcherBackend. setState( SparkAppHandle. State. RUNNING)
```

上面代码中的第 2 行构建了一个 Command 对象，该对象的第一个参数表示 mainClass，即进程的主类。该类在 Standalone 模式下为 org. apache. spark. executor. CoarseGrainedExecutorBackend。分别得到 sparkJavaopts、javaOpts、command、appUiAddress、coresPerExecutor 及 appDes 传入 AppClient 的构造函数。AppClient 将会向 Master 发送 Regis-

terApplication 注册请求，Master 受理后通过 launchExecutor 方法在 Worker 结点启动一个 ExecutorRunner 对象，该对象用于管理一个 Executor 进程。在 ExecutorRunner 中将通过 CommandUtil 构建一个 ProcessBuilder，调用 ProcessBuilder 的 start 方法将会以进程的方式启动 org. apache. spark. executor. CoarseGrainedExecutorBackend。在 CoarseGrainedExecotor-Backend 的 onStart 方法中，将会向 Driver 端发送 RegisterExecutor（executorId, self, host-Port, cores, extractLogUrls）消息请求注册，完成注册后将立即返回一个 RegisteredExecu-tor（executorAddress. host）消息，CoarseGraiendExecutorBackend 收到该消息，马上实例化出一个 Executor。源代码如下所示。

```
case RegisteredExecutor(hostname) =>
        logInfo("Successfully registered with driver")
        executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
```

从这里可以看出，CoarseGrainedExecutorBackend 比 Executor 先实例化。CoarseGrainedEx-ecutorBackend 负责与集群通信，而 Executor 则专注于任务的处理，它们是一对一的关系，在集群中各司其职，其关系如图 5-2 所示。

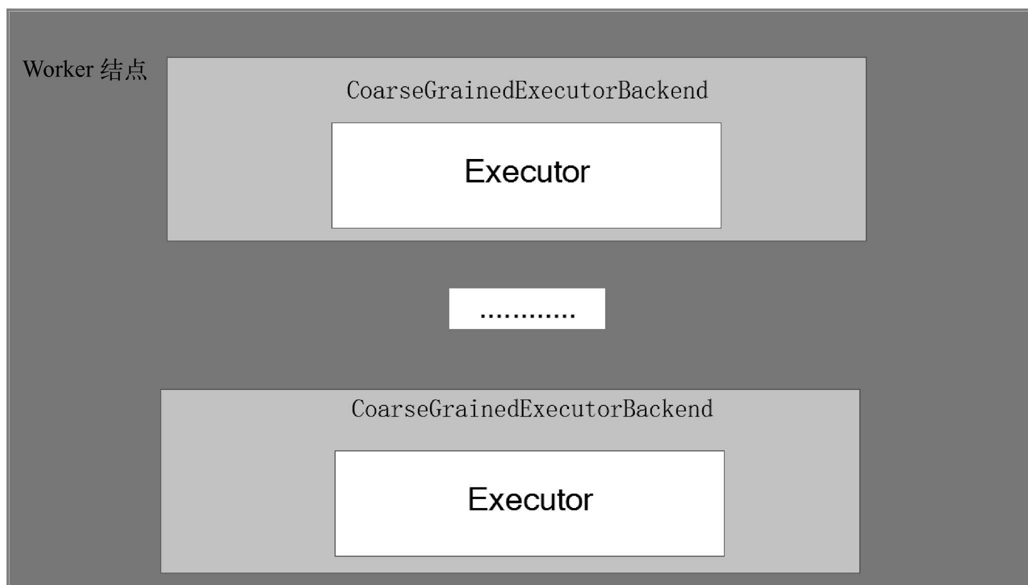


图 5-2　CoarseGrainedExecutorBackend 和 Executor

每一个 Worker 结点上可以启动多个 CoarseGrainedExecutorBackend 进程，每一个进程对应一个 Executor。

5.2.2 ExecutorBackend 的不同实现

ExecutorBackend 是一个和集群交互的接口，该接口在不同的调度模式下有不同的实现。图 5-3 所示是 ExecutorBackend 及其实现的关系类图。

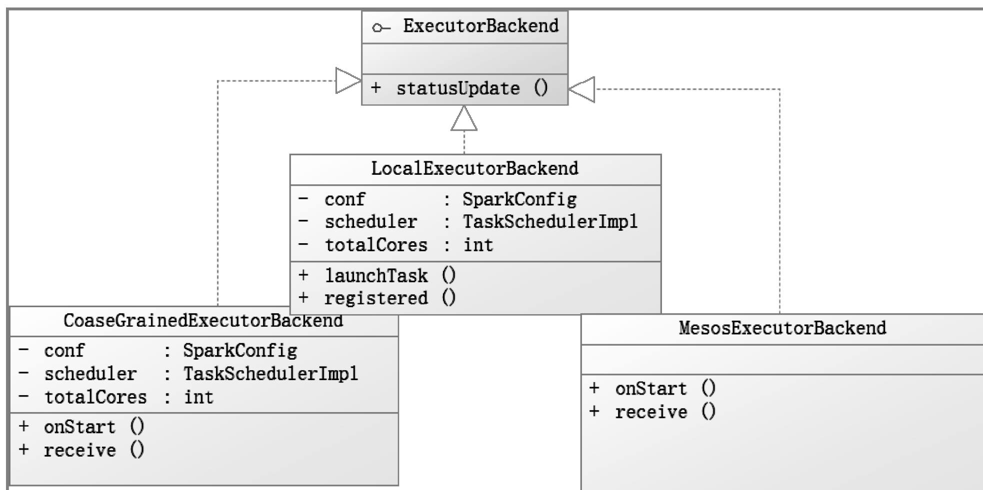


图 5-3　ExecutorBackend 的不同实现

不同模式下，ExecutorRunner 启动的进程是不一样的。在 Standalone 模式下，启动的是 org. apache. spark. executor. CoarseGrainedExecutorBackend 进程；在 Local 模式下启动的是 org. apache. spark. executor. LocalExecutorBackend 进程；在 Mesos 模式下启动的是 org. apache. spark. executor. MesosExecutorBackend 进程。

下面来看一下 Standalone 模式下 CoarseGrainedExecutorBackend 的启动。在 Standalone 模式下，会启动 org. apache. spark. deloy. Client 类，该类将向 Master 发送 RequestSubmitDriver (driverDescription) 消息，Master 中匹配到 RequestSubmitDriver(driverDescription)后，将会调用 schedule( )方法。该调用源代码如下所示。

```
case RequestSubmitDriver(description) => {
  //若 state 不为 ALIVE,直接向 Client 返回 SubmitDriverResponse(self,false,None,msg)消息
      if (state != RecoveryState.ALIVE) {
        val msg = s"${Utils.BACKUP_STANDALONE_MASTER_PREFIX}: $state. " +
          "Can only accept driver submissions inALIVE state."
        context.reply(SubmitDriverResponse(self,false,None,msg))
      } else {
        logInfo("Driver submitted " + description.command.mainClass)
        //使用 description 创建 Driver,该方法返回 DriverDescription
        val driver = createDriver(description)
        //持久化引擎持久化该 Driver,以便异常退出错误恢复
        persistenceEngine.addDriver(driver)
        //waitingDrivers 等待调度数组中加入该 Driver
        waitingDrivers += driver
        //向 drivers 这个 HashSet 中加入 Driver
        drivers.add(driver)
        //调用 schdule 方法调度资源
        schedule()
```

```
            //向 ClientEndpoint 回复 SubmitDriverResponse 消息
             context.reply(SubmitDriverResponse(self,true,Some(driver.id),
               s"Driver successfully submitted as ${driver.id}"))
          }
        }
```

```
   private def schedule():Unit = {
 //若 Master 的状态不为 ALIVE,则直接返回
     if (state != RecoveryState.ALIVE) { return }
     // Drivers take strict precedence over executors
   //随机打乱 Worker 这个 HashSet 中 Worker 的次序,用于 Driver 在集群中随机加载,起到平衡加载的作用,从这里也可以看出,Driver 是在集群中随机的 Worker 上启动的
     val shuffledWorkers = Random.shuffle(workers) // Randomization helps balance drivers
   //遍历状态为 ALIVE 的 Worker
     for (worker <- shuffledWorkers if worker.state == WorkerState.ALIVE) {
   //从 waitingDrivers 中取出 Driver,并进行匹配
       for (driver <- waitingDrivers) {
   //如果 Driver 和 Worker 匹配,内存和运行核都将得到满足
         if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {
   //在该 Worker 上启动加载 Driver
   launchDriver(worker,driver)
   //加载 Driver 后,从 waitingDrivers 中移除该 Driver
             waitingDrivers -= driver
           }
         }
       }
     startExecutorsOnWorkers()//为 application 在 worker 上启动 Executor
   }
```

上面代码中，RecoveryState 若不为 ALIVE，则直接返回，否则使用 Random.shuffle 将 Worker 集合打乱生成新的集合 shuffledWorkers，这样会尽量考虑到选择 Driver 的负载均衡。在 for 语句中遍历 shuffledWorkers，首先，若 Worker 的 WorkerStage 为 ALIVE，则遍历 waitingDrivers 队列，判断 Worker 剩余内存和剩余物理核是否满足 Driver 需求，如满足则调用 launchDriver（worker，driver）方法在选中的 Worker 上启动 Driver 进程。

Driver 在 Worker 结点上启动起来之后，会实例化 SparkContext，在 SparkContext 中将实例化出 DAGScheduler 和 SparkDeloySchedulerBackend。在 SparkDeploySchedulerBackend 中将会 new 一个 AppClient，AppClient 中有一个 ClientEndpoint，在其 onStart 方法中将向 Master 发送 RegisterApplication 请求注册 application，注册好 application 之后 Master 又会调用 schedule 方法。在满足条件的 Worker 上为 application 启动 Executor，首先会启动 ExecutorRunner，在 ExecutorRunner 中启动 CoarseGrainedExecutorBackend，启动后将会实例化出 Executor。为什么在 Standalone 模式下会启动 CoarseGrainedExecutorBackend 呢？是在什么地方设置要启动 Coar-

seGrainedExecutorBackend 进程的呢？其实在实例化 AppClient 时，就已经传入了，源代码如下所示。

```
1.  //command 中指定 ExecutorBackend 的实现类
2.  val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
3.    args,sc.executorEnvs,classPathEntries ++ testingClassPath,libraryPathEntries,javaOpts)
4.     val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
5.     val coresPerExecutor = conf.getOption("spark.executor.cores").map(_.toInt)
6.    //使用 Commond 等信息构建 application 的描述对象 ApplicationDescription
7.  val appDesc = new ApplicationDescription(sc.appName,maxCores,sc.executorMemory,
8.        command,appUIAddress,sc.eventLogDir,sc.eventLogCodec,coresPerExecutor)
9.  //将 appDesc 作为参数,创建 AppClient
10. client = new AppClient(sc.env.rpcEnv,masters,appDesc,this,conf)
11. //调用 start 方法,启动 AppClient
12.    client.start()
13. //将状态设置成为 SUBMITTED
14.    launcherBackend.setState(SparkAppHandle.State.SUBMITTED)
15. //等待登记注册
16.    waitForRegistration()
17. //将状态设置成为 RUNNING
18.    launcherBackend.setState(SparkAppHandle.State.RUNNING)
```

上面代码中的第 2 行设置了 Command 对象，Command 对象的第一个参数是启动进程的 mainClass。因此在 ExecutorRunner 中启动进程时，启动的即是 org.apache.spark.executor.CoarseGrainedExecutorBackend。

5.2.3 ExecutorBackend 中的通信

在 ExecutorBackend 中有 statusUpdate（taskId:Long,state:TaskState,data:ByteBuffer）方法，通过这个方法向集群发送 Task 执行的各种信息。如果任务执行失败，返回失败的信息；如果执行成功，返回任务执行的结果。在本节中，将重点讲解在 Standalone 模式下，CoarseGrainedExecutorBackend 中的通信。CoarseGrainedExecutorBackend 在整个集群中的通信如图 5-4 所示。

在图 5-4 中，Executor 与 CoarseGrainedExecutorBackend 协作，任务计算的结果通过 CoarseGrainedExecutorBackend 的 statusUpdate 方法将 taskId、TaskState 及结果数据发送给 Driver。Driver 收到 StatusUpdate（executorId,tasked,state,data）消息，通过判断 state 的不同的状态来进行不同的处理。比如，当 state 的状态为 TaskState.LOST 时，Driver 端将会移除 Executor；当 state 状态为 TaskState.FINISHED 时，Driver 端将会调用 enqueueSuccessfulTask 进行处理。

首先看一下 CoarseGrainedExecutorBackend 与 Driver 之间的通信。当在 Worker 结点中启动 ExecutorRunner 时，ExecutorRunner 中会启动 CoarseGrainedExecutorBackend 进程，在 Coar-

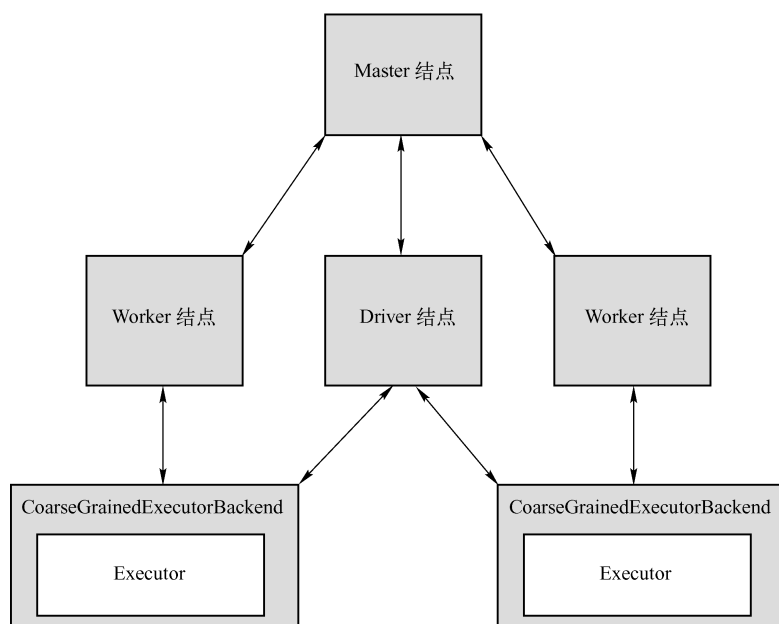


图 5-4　CoarseGrainedExecutorBackend 在整个集群中的通信

seGrainedExecutorBackend 的 onStart 方法中，向 Driver 发出 RegisterExecutor 注册请求。源代码如下所示。

```
override def onStart() {
    logInfo("Connecting to driver:" + driverUrl)
    rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
      // This is a very fast action so we can use "ThreadUtils.sameThread"
      driver = Some(ref)
     //向 Driver 发送 ask 请求,等待 Driver 的回应
      ref.ask[RegisterExecutorResponse](
        RegisterExecutor(executorId,self,hostPort,cores,extractLogUrls))
    }(ThreadUtils.sameThread).onComplete {
     //成功
      case Success(msg) => Utils.tryLogNonFatalError {
        Option(self).foreach(_.send(msg)) // msg must be RegisterExecutorResponse
      }
     //失败
      case Failure(e) => {
        logError(s"Cannot register with driver: $ driverUrl",e)
        System.exit(1)
      }
    }(ThreadUtils.sameThread)
  }
```

上面代码中，Some(ref)得到 Driver 的引用，通过 ask 方法返回 Future[RegisterExecutor-Response]，然后在 Future 对象上调用 onComplete 方法进行额外的处理。Driver 端收到注册请求，将会注册 Executor 的请求，并向 ListenerBus 中发送 SparkListenerExecutorAdded 事件。处理源代码如下所示。

```
case RegisterExecutor(executorId,executorRef,hostPort,cores,logUrls) =>
//检查 executorDataMap 中是否包含该 executorId,如果包含,返回 RegisterExecutorFailed 消息
  if (executorDataMap.contains(executorId)) {
      context.reply(RegisterExecutorFailed("Duplicate executor ID:" + executorId))
    } else {
      //若 executorRef.address 地址不为 null,取出 executorRef 的地址作为 executorAddress,否则使用 sender 的 Address 作为 executorAddress
      val executorAddress = if (executorRef.address != null) {
executorRef.address
        } else {
          context.senderAddress
        }
          logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executorId")
      //在 addressToExecutorId 这个哈希表中加入 executorAddress 和 executorId 的对应关系
    addressToExecutorId(executorAddress) = executorId
      //totalCore 增加 core 的个数
      totalCoreCount.addAndGet(cores)
      //totalRegisteredExecutors 加 1
      totalRegisteredExecutors.addAndGet(1)
      //创建 ExecutorData 对象
  val data = new ExecutorData(executorRef,executorRef.address,executorAddress.host,
        cores,cores,logUrls)
      //同步代码块
      CoarseGrainedSchedulerBackend.this.synchronized {
    //在 executorDataMap 中加入 executorId 和 ExecutorData 的对应关系
      executorDataMap.put(executorId,data)
        //如果挂起的 Executor 的数量大于 0
        if (numPendingExecutors > 0) {
        //挂起 Executor 的数量减 1
      numPendingExecutors -= 1
      logDebug(s"Decremented number of pending executors ($numPendingExecutors left)")
        }
      }
      //向 CoarseGrainedExecutorBackend 回复 RegisteredExecutor 消息
```

```
                context. reply(RegisteredExecutor(executorAddress. host))
                //向事件总线中发送 SparkListenerExecutorAdded 消息
                listenerBus. post(
                    SparkListenerExecutorAdded(System. currentTimeMillis(),executorId,data))
                //调用 makeOffers,给 Executor 发送执行任务
                makeOffers()
            }
```

如上面代码所示，如果 executorDataMap 中已经存在该 Executor 的 id，返回 RegisterExecutorFailed；如果不存在该 Executor 的 id，则在 executorDataMap 中加入该 Executor 的 id，并返回 RegisteredExecutor 消息且向 listenerBus 中添加 SparkListenerExecutorAdded 事件。CoarseGrainedExecutorBackend 收到 RegisteredExecutor 消息后，将会新建一个 Executor 执行器，并为此 Executor 充当信使与 Driver 通信。CoarseGrainedExecutorBackend 收到 RegisteredExecutor 消息的源代码如下所示。

```
case RegisteredExecutor(hostname) =>
        logInfo("Successfully registered with driver")
        //收到 RegisteredExecutor 消息,立即创建 Executor
executor = new Executor(executorId,hostname,env,userClassPath,isLocal = false)
```

从上面代码中可以看到，CoarseGrainedExecutorBackend 收到 RegisteredExecutor（hostname）消息后，将会新建一个 Executor。由此可见，Executor 较 CoarseGrainedExecutorBackend 后实例化，这与 Executor 和 CoarseGrainedExecutorBackend 的不同职责有关，Executor 主要负责计算，而 CoarseGrainedExecutorBackend 主要负责通信，通信环境准备好了，架起了同 CoarseGrainedSchedulerBackend 通信的桥梁，就可以接收 CoarseGrainedSchedulerBackend 中调用 launchTask 方法发送的 LaunchTask 消息了，因此通信在前，计算在后。

Executor 中的计算结果是通过 CoarseGrainedExecutorBackend 的 statusUpdate 方法返回给 CoarseGrainedExecutorBackend 的，statusUpdate 方法的代码如下所示。

```
override def statusUpdate(taskId:Long,state:TaskState,data:ByteBuffer) {
    //构建 StatusUpdate 对象
    val msg = StatusUpdate(executorId,taskId,state,data)
     driver match {
      //向 Driver 发送 StatusUpdate 消息
        case Some(driverRef) => driverRef. send(msg)
        case None => logWarning(s"Drop $ msg because has not yet connected to driver")
      }
   }
```

上面源代码中，通过参数 taskId、state 和 data 构建一个 StatusUpdate 对象，该对象将被当做消息发送到 Driver 端，Driver 根据返回结果的需要，将会向 CoarseGrainedExecutorBackend 发送新的指令消息，如 LaunchTask、KillTask、StopExecutors 和 Shutdown 等。

Section

5.3 执行器（Executor）中任务的执行

5.3.1 Executor 中任务的加载

Executor 是基于线程池的任务执行器，通过 launchTask( )方法加载任务，将任务以 TaskRunner 的形式放入线程池中运行。

DAGScheduler 划分好 Stage 并通过 submitMissingTasks 方法分配好任务，并把任务交由 TaskSchedulerImpl 的 submitTasks 方法，将任务加入调度池，之后调用 CoarseGrainedSchedulerBackend 的 riviveOffers 方法为 Task 分配资源指定 Executor。任务资源都分配好之后，CoarseGrainedSchedulerBackend 将向 CoarseGranedExecutorBackend 发送 LaunchTask 消息，将具体的任务发送到 Executor 上进行计算。

CoarseGranedExecutorBackend 匹配到 LaunchTask （data） 消息之后，将会调用 Executor 的 launchTask 方法。launchTask 方法中将会构建 TaskRunner 对象并放入线程池中执行。

具体的调用流程如图 5-5 所示。

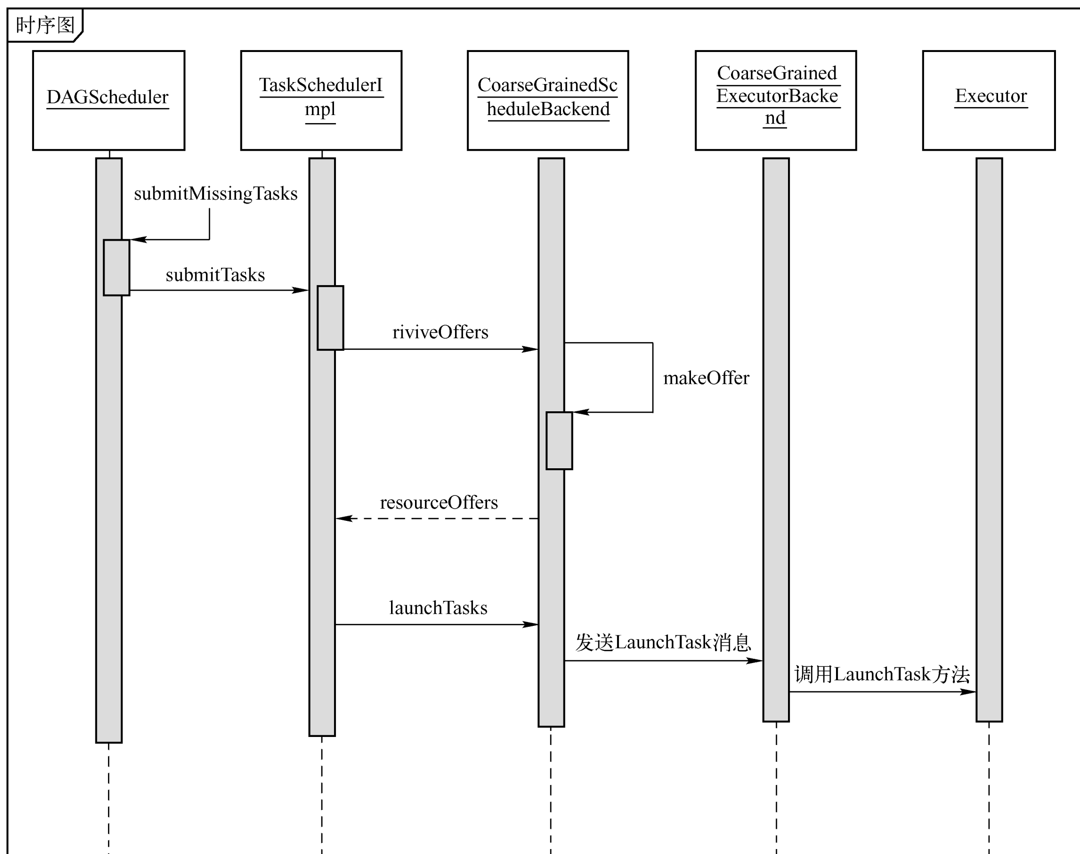


图 5-5 Executor 中 Task 的加载时序图

任务加载好了之后，在 Executor 中将会把构建好的 TaskRunner 放入线程池运行，至此任务完成加载，开始运行。

5.3.2 Executor 中的任务线程池

Executor 是构建在线程池之上的任务执行器。在 Executor 中使用线程池可以减少在创建和销毁线程上所花的时间和系统资源开销。如果不使用线程池，可能造成系统创建大量的线程而导致消耗完系统内存，以及出现“过度切换”。

在 Executor 中使用线程池基于以下两点原因，首先在 Executor 端执行的任务处理时间都比较短，需要频繁地创建和销毁线程，这样就带来了巨大的创建和销毁线程的开销，造成额外的系统资源开销；其次 Executor 中处理的任务数量巨大，如果每一个任务都创建一个线程，将导致消耗完系统内存，出现“过度切换”。

首先看一下 Executor 中的线程池。Executor 中使用的是 CachedThreadPool，使用这种类型线程池的好处是，任务比较多时可以自动新增处理线程，而任务比较闲时自动回收空闲线程。Executor 中的线程池的源代码定义如下所示。

```
private val threadPool = ThreadUtils.newDaemonCachedThreadPool("Executor task launch worker")
```

CoarseGrainedExecutorBackend 调用 Executor 的 launchTask 方法，在方法中将会新建 TaskRunner，然后放入线程池进行处理。源代码如下。

```
1. val tr = new TaskRunner(context,taskId = taskId,attemptNumber = attemptNumber,taskName,
2.             serializedTask)
3.       runningTasks.put(taskId,tr)
4. threadPool.execute(tr)
```

从上面的源代码中可以看到，新建的 TaskRunner 对象首先放入 runningTasks 这样一个 ConcurrentHashMap 里面，然后使用线程池的 execute 方法运行 TaskRunner，execute 方法将会调用 TaskRunner 的 run 方法，在 TaskRunner 的 run 方法中执行计算任务。

5.3.3 任务执行失败处理

TaskRunner 在计算的过程中可能发生各种异常甚至错误，例如抓取 Shuffle 结果失败、任务被杀死、没权限向 HDFS 写入数据等，当 TaskRunner 的 run 方法运行时，可以通过 try catch 语句捕获这些异常，并通过调用 CoarseGrainedExecutorBackend 的 statusUpdate 方法向 CoarseGrainedSchedulerBackend 汇报。

下面是 CoarseGrainedExecutorBackend 的 statusUpate 方法源代码。

```
1. override def statusUpdate(taskId:Long,state:TaskState,data:ByteBuffer){
2.        val msg = StatusUpdate(executorId,taskId,state,data)//构建 StatusUpdate 消息对象
3.       driver match {
```

```
        case Some(driverRef) => driverRef.send(msg)//向 driver 发送 StatusUpdate 消息
        case None => logWarning(s"Drop $ msg because has not yet connected to driver")
    }
}
```

在 statusUpdate 方法中，通过方法的参数 taskId、state 和 data 构建一个 StatusUpdate 对象，并通过 driverRef 的 send 方法将该对象发送回 CoarseGrainedSheduleBackend。CoarseGrainedScheduleBackend 匹配到 StatusUpdate 时，将根据 StatusUpdate 对象中的 state 值来对该 Task 的执行情况做出判断，并执行不同的处理逻辑。

TaskState 是一个枚举变量，其中包括 LAUNCHING、RUNNING、FINISHED、FAILED、KILLED 和 LOST 这些枚举值。Executor 根据任务执行的不同状态，通过 statusUpdate 方法返回特定的 TaskState 值，该值通过 ExecutorBackend 返回给 SchedulerBackend，在 SchedulerBackend 中根据 TaskState 中的值进行处理。

先来看一下 TaskState. FAILED 这种情况，在 Executor 的 run 方法中，如果发生 FetchFailedExeception、CommitDeniedExeception 或者其他 Throwable 的子类的异常，都将会返回 TaskState. FAILED 状态，该状态通过 CoaseGainedExecutorBackend 返回。在 CoaseGaiendSchedulerBackend 中，匹配到 StatusUpdate 消息后将进行相应的处理，匹配代码如下。

```
case StatusUpdate(executorId,taskId,state,data) =>
        scheduler.statusUpdate(taskId,state,data.value)//调用 TaskSchedulerImpl 更新状态
        if(TaskState.isFinished(state)){//若状态为 FINISH,则从 executorDataMap 中取出 executorId 对应的 ExecutorInfo
          executorDataMap.get(executorId) match {
            case Some(executorInfo) =>
              executorInfo.freeCores += scheduler.CPUS_PER_TASK
              makeOffers(executorId)//makeOffers 方法重新分配资源
            case None =>
              //Ignoring the update since we don't know about the executor.
              logWarning(s"Ignored task status update( $ taskId state $ state)" +
                s"from unknown executor with ID $ executorId")
          }
  }
```

上面代码中，首先调用 TaskSchedulerImpl 的 statusUpdate 方法，该方法的调用用于更新 taskId 对应任务的状态。完成更新后，判断 state 状态是否为 finished，若状态为 finished，则从 executorDataMap 这个哈希表中取出 executorId 对应的 ExecutorData 对象，修改该对象中的 freeCores。因为状态已经为 finished 状态，因此 ExecutorData 中的 freeCores 会增加 CPUS_PER_TASK 个，CPU_PER_TASK 为每一个任务占用的 CPU 核的个数，该个数可以通过 spark. task. cpus 配置项进行配置。

更新完成 ExecutorData 上的可用 CPU 后，这些闲置的 CPU 通过 makeOffers 方法再次分配给其他任务使用。先来看一下 makeOffers 方法的源代码，如下所示。

```
// Make fake resource offers on just one executor
    private def makeOffers(executorId:String) {
      //过滤存活的 Executor
      if(executorIsAlive(executorId)) {
        //从 executorDataMap 这个哈希表中取出 executorId 对应的 ExecutorData 对象,ExecutorData 表示 Executor 上的一组资源
        val executorData = executorDataMap(executorId)
        //使用 executorData 创建 WorkerOffer 对象,该对象代表 Executor 上可用的资源
        val workOffers = Seq(
          newWorkerOffer(executorId,executorData.executorHost,executorData.freeCores))
          //调用 TaskSchedulerImpl 上的 resourceOffers 方法,为任务分配运行资源,该方法返回获得运行资源的任务集合,之后运行 launchTasks 方法,将这些任务发送到 Executor 上运行
    launchTasks(scheduler.resourceOffers(workOffers))
      }
    }
```

每一个 Executor 上的资源发生变动时，都将调用 makeOffers 方法，该方法的作用是为等待执行的任务分配资源，并通过 launchTasks 方法将这些任务发送到这些 Executor 上运行。这些任务将被包装成 TaskRunner 对象，运行于 Executor 上的线程池中。

5.3.4 剖析 TaskRunner

TaskRunner 位于 Executor 中，继承自 Runnable 接口，代表一个可执行的任务，Driver 端下发的任务最终都要在 Executor 中封装成 TaskRunner，在 TaskRunner 的 run 方法中，将会进行任务的解析并调用 Task 接口的 run 方法进行计算计算。TaskRunner 的定义代码如下。

```
class TaskRunner(
    execBackend:ExecutorBackend,//通过 execBackend 和 SchedulerBakend 通信
    val taskId:Long,
    val attemptNumber:Int,
    taskName:String,
    serializedTask:ByteBuffer)
  extends Runnable {
```

TaskRunner 的构造函数中有 execBackend、taskId、attemptNumber、taskName 和 serializedTask 共 5 个参数。其中 execBackend 作为和 CoarseGrainedSchedulerBackend 通信的使者传入到 TaskRunner 中，在任务计算状态发生变化时，调用 execBackend 的 statusUpdate 方法向 CoarseGrainedSchedulerBackend 报告。传入 taskId 是为了使用 TaskMemoryManager 管理该 Task。attemptNumber 代表任务尝试执行的次数，serializedTask 是序列化的任务，序列化的任务通过序列化工具反序列化得到任务对象。

那么，在 TaskRunner 中是如何运行任务的呢？大家知道，在线程池中启动 Runnable 任

务会自动调用 Runnable 的 run 方法，TaskRunner 作为一个 Runnable 接口的实现类，启动时会自动调用其 run 方法。run 方法中主要完成以下任务。

- 调用 ExecutorBackend 的 statusUpdate 方法向 SchedulerBackend 发送任务状态更新消息。
- 反序列化出 Task 和相关依赖 Jar 包。
- 调用 Task 上的 run 方法运行任务。
- 返回 Task 运行结果。

Task 是一个接口，ResultTask 和 ShffleMapTask 是它的两种实现。Task 接口中提供了 run 方法，用于运行任务。在 TaskRunner 的 run 方法中，会通过反序列化器反序列化出 Task 并调用 Task 上的 run 方法运行任务，这里怎么知道是 ResultTask 还是 ShffleMapTask 呢？其实不管是 ResultTask 还是 ShffleMapTask 都一视同仁，因为 ResultTask 和 ShffleMapTask 都实现了 Task 接口，都有 run 方法。这正是面向接口编程带来的最大的好处，即灵活且最大限度地复用代码。

Task 将运行结果分为 3 种情况来处理，第一种情况是 resultSize 大于 maxResultSize，这种情况下构建 IndirectTaskResult 对象，并返回该 IndirectTaskResult 对象，IndirectTaskResult 对象中包含结果所在的 BlockId，在 SchedulerBackend 中可以通过 BlockManager 获得该 BlockId 对应的结果数据，这里的 maxResultSize 默认为 1 GB；第二种情况是 resultSize 大于 Akka 帧的大小，这种情况下也是构建 IndirectTaskResult 对象，并返回该 IndirectTaskResult 对象，Akka 帧的大小为 128 MB；第三种情况是直接返回 DirectTaskResult，这是在 resultSize 小于 Akka 帧大小的情况下采取的默认返回方式。

Section 5.4 小结

本章主要讲解了 Executor 及 ExecutorBackend 接口。在本章的 5.1 小节中，通过一个时序图从整体上把握 Executor 的启动过程，从源代码的角度讲解了 Executor 创建的整个过程，并分别讲解了 Executor 中的资源分配及 Executor 的启动过程，还讲解了 Executor 运行过程中的异常处理。

Executor 只是一个任务的执行器，任务被下发到 Executor 中，首先包装成 TaskRunner 对象，然后放入到 Executor 中的线程池中运行。对于 Executor 与 Driver 端的通信，就要借助于 ExecutorBackend 接口了。本章 5.2 小节中讲解了 ExecutorBackend 的通信接口，讲解了 ExecutorBackend 与 Executor 的关系（其实它们是相互扶持的“两兄弟”，一个负者通信，另一个负者任务的执行）。在这一节中，还介绍了 ExecutorBackend 接口的不同实现，以及 ExecutorBackend 与 SchedulerBackend 通信的细节。

5.3 小节则着重讲解了 Executor 中的任务执行的细节，这些细节包括任务的加载，任务线程池，任务执行失败处理，以及任务的载体 TaskRunner。



第 6 章　Spark 的存储模块（Storage）

Spark Storage 模块为用户提供了高层次的抽象，将接口和实现进行了有效的隔离，使用户无须关心底层的实现，而将主要精力放在业务的实现上。第 5 章讲解了执行器（Executor），可知任务以 TaskRunner 的形式运行于线程池中，那么任务运行的结果数据存放在哪里，如何读取呢？这是由 Storage 模块来管理和实现的，在本章中，将会结合源代码详细分析 Spark 的 Storage 模块。

本章将讲解 Storage 模块的基本定义和接口。在此基础上，详细剖析 Storage 的整体架构。最后讲解 OFF_HEAP 的实现 Tachyon，介绍 Tachyon 在 Spark 中的运用及 API 的使用。

Section 6.1　Storage 概述

6.1.1　Storage 的概念

在 Spark 中存储模块被抽象成 Storage，顾名思义，Storage 是存储的意思，代表着 Spark 中的数据存储系统，负责管理和实现数据块（Block）的存放。其中存取数据的最小单元是 Block，数据由不同的 Block 组成，所有操作都是以 Block 为单位进行的。本质上讲 RDD 中的 Partition 和 Storage 中的 Block 是等价的，只是所处的模块不同看待的角度不一样而已。

Storage 抽象模块的实现分为两个层次，如图 6-1 所示。

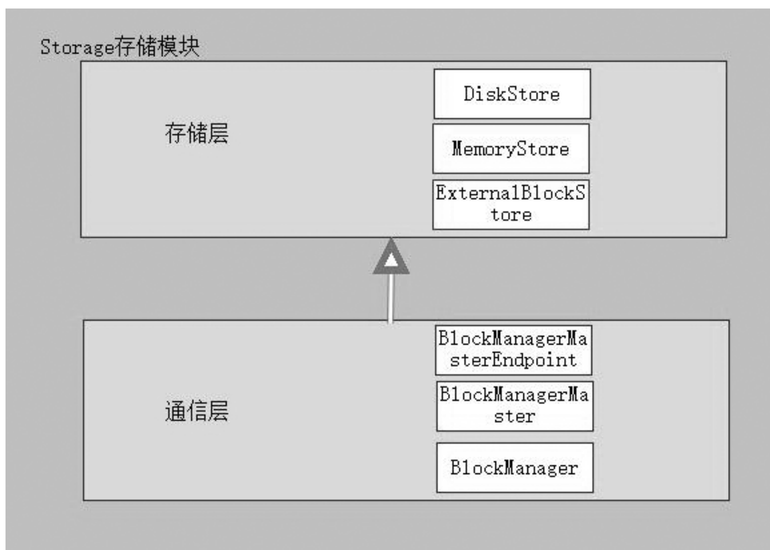


图 6-1　Storage 存储模块

1）通信层：通信层是典型的 Master – Slave 结构，Master 和 Slave 之间传输控制和状态信息。通信层主要由 BlockManager、BlockManagerMaster、BlockManagerMasterEndpoint、BlockManagerSlaveEndpoint 等类实现。

2）存储层：负者把数据存储到内存、磁盘或者堆外内存中，有时还需要为数据在远程结点上生成副本，这些都由存储层提供的接口实现。具体的存储层的实现类有抽象类 BlockStore，实现类 DiskStore、MemoryStore、ExternalBlockStore 等。

其他模块若要和 Storage 模块进行交互，需要通过调用统一的操作类 BlockManager 来完成。如果把整个存储模块看成一个黑盒，BlockManager 就是黑盒上留出的一个供外部调用的接口。黑盒中用到了不少的设计模式，下一小节将谈谈 Storage 抽象实现中涉及的设计模式。

6. 1. 2 Storage 的设计模式

Storage 模块中，对于存储层进行了抽象，使用接口将抽象和实现进行了完美的分离。对于块的存储，抽象出 BlockStore 类，在 BlockStore 类中提供抽象和钩子方法，实现不同存储时，就可以灵活的根据现实情况进行实现。BlockStore 有三个不同的实现，分别是 DiskStore、MemoryStore、ExternalBlockStore。在 BlockManager 中，存储块数据会根据 StorageLevel 的设置，选择 BlockStore 的不同实现，而 StorageLevel 则提供给用户，使用户可以根据机器性能、集群配置、使用场景等灵活的选择 block 的存储级别。

正是因为 BlockStore 的抽象，使得 BlockManager 的实现只需要关注 BlockStore 接口，而不必关心繁琐的实现细节，这是面向接口编程的典型应用。并且 Spark 为用户提供了详尽的 StorageLevel，用户无需自己定义，只需要根据实际情况灵活选择即可。

实际上 StorageLevel 是 Spark 为用户提供的一个存储功能选择开关，而实现的细节完全由 Spark Storage 模块隐藏。而且 Storage 模块对外提供了一个交互的接口 BlockManager，对于不同的存储如 Disk、Memory、External 等抽象出了 BlockStore，这使得 BlockManager 可以以面向接口编程的方式来使用不同的 Store，而无需关心具体 Store 的实现细节。通过面向接口编程，保证了模块代码的可复用性、灵活性、可扩展性以及程序的健壮性。

在 Storage 通信层里面，Slave 结点上的 Executor 中的 BlockManager 通过 BlockManagerSlaveEndPoint 和 Master 结点的 BlockManagerMasterEndpoint 进行通信。RPC 远程过程调用在 Spark 中担任通信功能，大大简化了 Spark 架构复杂度。下一小节将详细讲解 Storage 模块的整体架构。

Section 6. 2 Storage 模块整体架构

本小节主要面向源代码讲解 Storage 模块通信层及存储层的架构。6. 2. 1 小节讲解 Storage 模块的通信层，6. 2. 2 小节讲解 Storage 模块的存储层。通过这两个小节的讲解，站在源代码的角度，详细剖析 Storage 整体的架构，使读者对整个存储模块有个详细的了解。

6.2.1 通信层

在 Storage 模块中，使用 RPC 框架进行通信。Storage 模块对外提供了一个统一的交互类 BlockManager。BlockManager 在每一个结点（包括 Driver 端和 Slave 端）都有创建。Slave 端创建的 BlockManager 将在 initialize 方法中向 Driver 端的 BlockManagerMasterEndpoint 发送 RegisterBlockManager（blockManagerId，maxMemSize，slaveEndpoint）消息，收到消息后，完成 Slave 端 BlockManager 在 Driver 端 BlockManager 的注册。如图 6-2 所示。

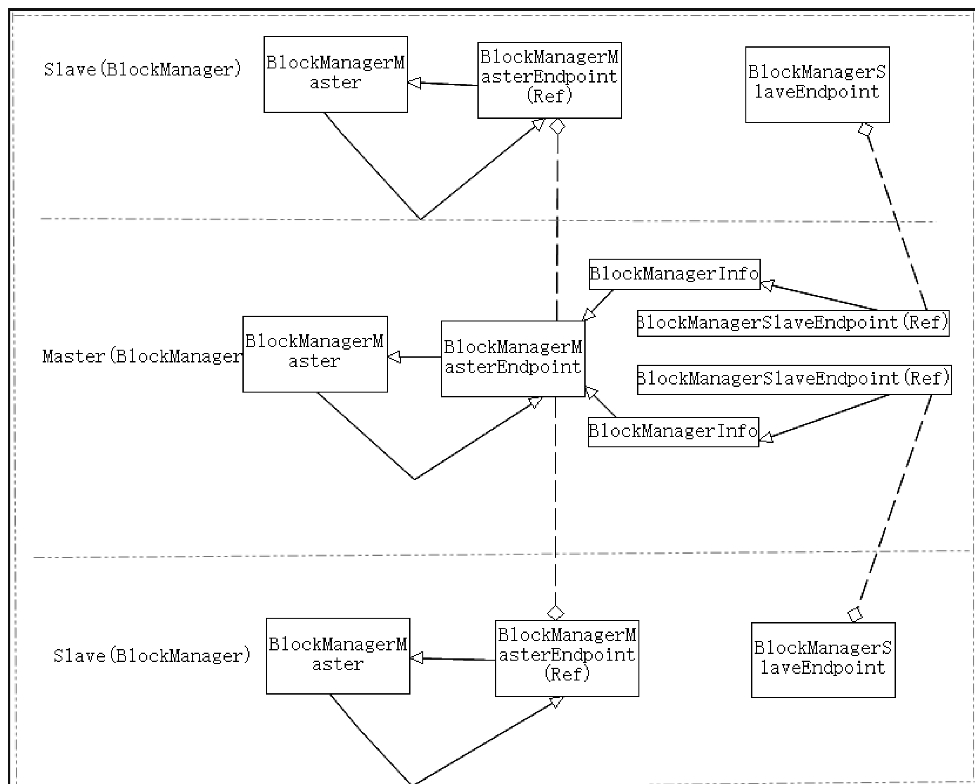


图 6-2　Storage 模块通信层

Driver 端的 BlockManagerMaster 拥有所有结点 BlockManagerSlaveEndpoint 的 Ref 及 BlockManagerMasterEndpoint 的引用。Slave 端的 BlockManagerMaster 拥有本结点上所有 BlockManagerSlaveEndpoint 的 Ref。Driver 和各结点通过 BlockManagerMasterEndpoint 和 BlockManagerSlaveEndpoint 通信。

在 Driver 和 Slaves 结点上的 Executor 中都有 BlockManager。BlockManager 提供了本地和远程不同存储类型（Disk、Memory、ExternalBlockStore）存储读取数据的接口。

首先来看一下 BlockManager 在 Driver 端的创建。在 SparkContext 创建时会根据具体的配置创建 SparkEnv 对象。源代码如下所示。

```
private[spark] def createSparkEnv(
        conf:SparkConf,
        isLocal:Boolean,
        listenerBus:LiveListenerBus):SparkEnv = {
    //创建 Driver 端的运行环境
SparkEnv.createDriverEnv(conf,isLocal,listenerBus,SparkContext.numDriverCores(master))
}
```

在 createSparkEnv 方法中传入 SparkConf 配置对象、isLocal 标志及 LiveListenerBus，方法中使用 SparkEnv 对象的 createDriverEnv 方法创建 SparkEnv 并返回。在 SparkEnv 的 createDriverEvn 方法中，将会创建 BlockManager、BlockManagerMaster 等对象，完成 Storage 在 Driver 端的部署。

SparkEnv 中创建 BlockManager 和 BlockManagerMaster 的关键源代码如下所示。

```
1.  //创建 BlockManagerMaster
2.  val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(
3.  BlockManagerMaster.DRIVER_ENDPOINT_NAME,
4.  //创建 BlockManagerMasterEndpoint
5.  new BlockManagerMasterEndpoint(rpcEnv,isLocal,conf,listenerBus)),
6.  conf,isDriver)
7.  //创建 BlockManager
8.  val blockManager = new BlockManager(executorId,rpcEnv,blockManagerMaster,
9.  serializer,conf,memoryManager,mapOutputTracker,shuffleManager,
10. blockTransferService,securityManager,numUsableCores)
```

上面代码的第 2 行使用 new 关键字实例化出 BlockManagerMaster，并在代码第 8 行传入 BlockManager 的构造函数，实例化出 BlockManager 对象。这里的 BlockManagerMaster 和 BlockManager 属于聚合关系。BlockManager 主要对外提供统一的访问接口，BlockManager-Master 主要对内提供各结点之间的指令通信服务。

BlockManagerMaster 在 Driver 端和 Executors 中的创建稍有差别。首先来看在 Driver 端创建的情形。创建 BlockManagerMaster 传入的 isDriver 参数，isDriver 为 true 表示在 Driver 端创建，否则视为在 Slave 结点上创建。

当 SparkContext 中执行_env.blockManager.initialize(_applicationId)代码时，会调用 Driver 端 BlockManager 的 initialize 方法。initialize 方法的源代码如下所示。

```
def initialize(appId:String):Unit = {
    //调用 blockTransferService 的 initialize 方法,blockTransferService 用于在不同结点 fetch 数据,传送数据
    blockTransferService.init(this)
    //shuffleClient 用于读取其他 Executor 上的 shuffle files
    shuffleClient.init(appId)
    //得到 blockManagerId
```

```
    blockManagerId = BlockManagerId(
      executorId,blockTransferService. hostName,blockTransferService. port)
    //得到 shuffleServerId
    shuffleServerId = if(externalShuffleServiceEnabled) {
      logInfo(s"external shuffle service port = $ externalShuffleServicePort")
    BlockManagerId(executorId,blockTransferService. hostName,externalShuffleServicePort)
    } else {
      blockManagerId
    }
    //向 blockManagerMaster 注册 BlockManager。在 registerBlockManager 方法中传入了
  slaveEndpoint,slaveEndpoint 为 BlockManager 中的 RPC 对象,用于与 blockManagerMaster 进行
  通信
    master. registerBlockManager(blockManagerId,maxMemory,slaveEndpoint)
    //注册 shuffleServer
      DWif(externalShuffleServiceEnabled &&! blockManagerId. isDriver) {
      registerWithExternalShuffleServer()
    }
  }
```

如上面的源代码所示，initialize 方法使用 appId 初始化 BlockManager。主要完成以下几项任务。

1）初始化 BlockTransferService。

2）初始化 ShuffleClient。

3）创建 BlockManagerId。

4）将 BlockManager 注册到 BlockManagerMaster 上。

5）若 ShuffleService 可用，注册 ShuffleService。

在 BlockManager 的 initialize 方法上右击 Find Usages，可以看到 initialize 方法在两个地方得到调用，一个是 SparkContext，另一个是 Executor。在启动 Executor 时，会调用 BlockManager 的 initialize 方法。Executor 中调用 initialize 方法的源代码如下所示。

```
//CoarseGrainedExecutorBackend 中实例化 Executor,isLocal 设置成 false,即 Executor 中的 isLo-
cal 始终为 fasle
if(! isLocal) {
//向度量系统注册
env. metricsSystem. registerSource(executorSource)
//调用 BlockManager 的 initialize 方法,如前所述,initialize 方法将向 BlockManagerMaster 注
册,完成 Executor 中的 BlockManager 向 Driver 中的 BlockManager 的注册
env. blockManager. initialize(conf. getAppId)
}
```

上面代码中，调用了 env. blockManager. initialize 方法。在 initialize 方法中，完成 BlockManger 向 Master 端 BlockManagerMaster 的注册。使用方法 master. registerBlockManager(block-



ManagerId,maxMemory,slaveEndpoint)完成注册，registerBlockManager 方法中传入 blockManagerId、maxMemory 和 salveEndPoint 引用，分别表示 Executor 中的 BlockManager、最大内存和 BlockManger 中的 BlockMangarSlaveEndpoint。BlockManagerSlaveEndpoint 是一个 RPC 端点，通过它完成与 BlockManagerMaster 的通信。BlockManager 收到注册请求后将 Executor 中注册的 BlockManagerInfo 存入哈希表中，以便通过 BlockManagerSlaveEndpoint 向 Executor 发送控制命令。

6.2.2 存储层

Storage 的存储由统一接口 BlockStore 定义，BlockStore 是一个抽象类，定义了数据的读写方法。BlockStore 抽象类有不同的实现，目前有 DiskStore、MemoryStore、ExternalBlockStore 三种不同的实现。下面是 BlockStore 抽象接口的定义。

```
private[spark] abstract class BlockStore(val blockManager:BlockManager)extends Logging {
//通过指定的 StorageLevel,存入字节数组
def putBytes(blockId:BlockId,bytes:ByteBuffer,level:StorageLevel):PutResult
//通过指定的 StorageLevel,存入任意类型的的数据 values,前提是 values 是可迭代的,returnValues 用于指定是否返回值
def putIterator(blockId:BlockId,values:Iterator[Any],level:StorageLevel,returnValues:Boolean):PutResult
//通过指定的 StorageLevel,存入任意类型的数据,这些数据是放入数组中的,returnValues 用于指定是否返回值
def putArray(blockId:BlockId,values:Array[Any],level:StorageLevel,returnValues:Boolean):PutResult
//得到 blockId 对应的数据集合的大小
  def getSize(blockId:BlockId):Long
  //得到 blockId 对应的数据集合的数据,返回类型是 Option[ByteBuffer]
  def getBytes(blockId:BlockId):Option[ByteBuffer]
  //得到 blockId 对应的数据,返回类型是 Iterator[Any]
  def getValues(blockId:BlockId):Option[Iterator[Any]]
  //移除 blockId 对应的数据,移除成功返回 true
  def remove(blockId:BlockId):Boolean
  //DiskStore 中是否包含 blockId 对应的数据
  def contains(blockId:BlockId):Boolean
  //用于清除数据
  def clear(){}
}
```

上面源代码是 BlockStore 抽象类的定义。putBytes、putIterator、putArray 三个接口方法类似，都是按照指定的存储级别存储集合字节信息，getSize 方法返回指定 blockId 对应数据块的长度，getBytes 返回指定 blockId 对应的数据块的字节数组。remove 方法移除指定 blockId 对应的数据，若该 blockId 存在并且移除成功，返回 true，否则返回 false。Contains 方法用于

判断 blockId 对应的数据是否存在。clear 方法是一个钩子方法，在 DiskStore 和 ExternalBlockStore 中不做任何处理，在 MemoryStore 中用于清除内存中的 MemoryEntry，用于释放内存空间。

Storage 存储按 BlockStore 的实现分为三种情况：

1）DiskStore：存储数据到磁盘。

2）MemoryStore：以字节数组或 Java 对象的方式存储在内存中。

3）ExternalBlockStore：存储数据到 OFF_HEAP 中，Spark 目前的 OFF_HEAP 默认实现采用 Tachyon。

BlockStore 是存储模块的抽象，提供了存取数据的基本的抽象方法。DiskStore、MemoryStore、ExternalBlockStore 都实现 BlockStore 接口，它们的继承层次类图如图 6-3 所示。

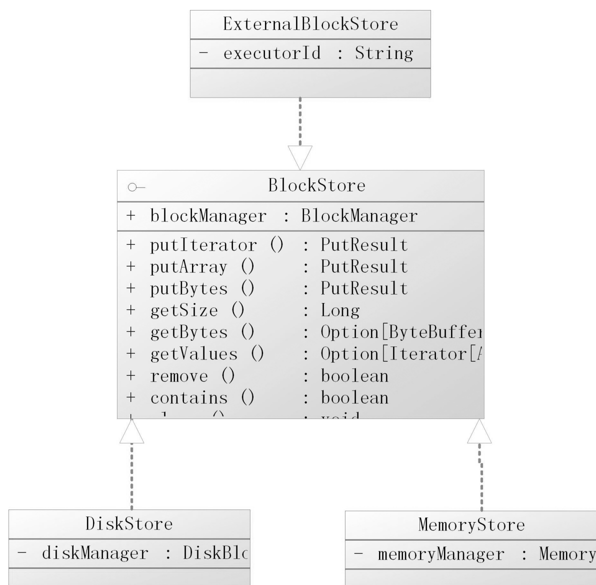


图 6-3 BlockStore 的继承结构

使用 BlockStore 时，可以指定 StorageLevel，StorageLevel 中存放的是一些标志信息，这些标志信息用于判断 RDD 数据进行何种方式的存储。下面是 StorageLevel 的部分源代码。

```
class StorageLevel private(
//是否使用磁盘
private var_useDisk:Boolean,
//是否使用内存
private var_useMemory:Boolean,
//是否使用 OFF_HEAP
private var_useOffHeap:Boolean,
//是否序列化
private var_deserialized:Boolean,
//副本个数,默认为 1
```




```
private var_replication:Int = 1)
extends Externalizable {
//下面是 Spark 中一些预定义好的存储级别,这些存储级别可以满足几乎所有场景需求
//不设置存储级别
val NONE = new StorageLevel(false,false,false,false)
//只是用磁盘
  val DISK_ONLY = new StorageLevel(true,false,false,false)
//只是用磁盘,并且副本个数设置为 2
  val DISK_ONLY_2 = new StorageLevel(true,false,false,false,2)
//只是用内存
  val MEMORY_ONLY = new StorageLevel(false,true,false,true)
//只是用内存,副本个数为 2
  val MEMORY_ONLY_2 = new StorageLevel(false,true,false,true,2)
//只是用内存并且序列化数据,这些数据以字节数组的形式存放于内存中,大大减小内存中对象的个数和占用内存的空间,但是需要额外的序列化和反序列化性能开销
  val MEMORY_ONLY_SER = new StorageLevel(false,true,false,false)
//只是用内存并且序列化,设置副本个数为 2。这些数据以字节数组的形式存放于内存中,大大减小内存中对象的个数和占用内存的空间,但是需要额外的序列化和反序列化性能开销
  val MEMORY_ONLY_SER_2 = new StorageLevel(false,true,false,false,2)
//使用内存和磁盘进行存储,Spark Executor 中默认用于存储的内存的大小为 Executor 分配内存的 3/8,超出该设置的大小,数据将外溢到磁盘
  val MEMORY_AND_DISK = new StorageLevel(true,true,false,true)
//使用内存和磁盘进行存储,设置副本个数为 2。Spark Executor 中默认用于存储的内存的大小为 Executor 分配内存的 3/8,超出该设置的大小,数据将外溢到磁盘
  val MEMORY_AND_DISK_2 = new StorageLevel(true,true,false,true,2)
//使用内存和磁盘并序列化存储。Spark Executor 中默认用于存储的内存的大小为 Executor 分配内存的 3/8,超出该设置的大小,数据将外溢到磁盘。
  val MEMORY_AND_DISK_SER = new StorageLevel(true,true,false,false)
//使用内存和磁盘并序列化存储,副本个数为 2。Spark Executor 中默认用于存储的内存的大小为 Executor 分配内存的 3/8,超出该设置的大小,数据将外溢到磁盘
  val MEMORY_AND_DISK_SER_2 = new StorageLevel(true,true,false,false,2)
//使用 OFF_HEAP,Spark 中目前使用 Tachyon 实现 OFF_HEAP 存储
  val OFF_HEAP = new StorageLevel(false,false,true,false)
```

StorageLevel 的几个构造参数是_useDisk、_useMemory、_useOffHeap、_deserialized、_replication。在构建 StorageLevel 的时候，根据传入参数的不同，实例化出不同的 StorageLevel。在 Spark 中已经预先定义出了很多 StorageLevel，例如 DISK_ONLY、MEMORY_ONLY 等，这些预定义的存储级别可以直接通过 StorageLevel 对象引用，并且这些预定义的存储级别几乎能满足所有存储情景。

下面分别对 BlockStore 的三种不同实现：DiskStore、MemoryStore、ExternalBlockStore 存储数据进行讲解。

1. DiskStore

顾名思义，DiskStore 会将 Block 存放到磁盘上，要存储数据就需要指定数据存放的路径，怎么设置存放 Block 的路径呢？在 DiskStore 中可以配置多个存放 Block 的目录，DiskBlockManger 会根据这些配置创建不同的文件夹，存放 Block。文件夹的命名格式为 prefix－UUID 的形式，Prefix 为指定的文件名前缀（默认为 blockmgr），UUID 为 UUID 算法生成的字符串，Block 将会存放在所有的创建目录中并保存副本。DiskBlockManger 会调用 createLocalDirs 方法为 Block 创建文件夹，源代码如下。

```
private def createLocalDirs(conf:SparkConf):Array[File] = {
    //从 SparkConf 配置对象中找出配置的文件保存路径,配置项为 spark.local.dir,路径可以配置多个,用逗号分隔开
    Utils.getConfiguredLocalDirs(conf).flatMap { rootDir =>
        try {
     //调用 Utils 的 createDirectory 方法,创建目录,该方法返回创建好的目录 localDir,该方法接受两个参数,第一个参数是要创建的目录的路径,第二个参数是创建目录的前缀
            val localDir = Utils.createDirectory(rootDir,"blockmgr")
            logInfo(s"Created local directory at $localDir")
            //返回创建好的目录
            Some(localDir)
          } catch {
            case e:IOException =>
              logError(s"Failed to create local dir in $rootDir. Ignoring this directory. ",e)
              None
          }
      }
  }
```

上面代码中，传入 SparkConf，通过 Utils 得到 conf 中配置的多个文件的根目录，并使用 Utils.createDirectory(rootDir,"blockmgr")方法实际创建根目录，这些目录可以通过 spark.local.dir 配置项配置。下面是 Utils 的 createDirectory 方法的源代码。

```
//该方法在给定的 root 目录下创建子目录
def createDirectory(root:String,namePrefix:String = "spark"):File = {
   //定义变量,表示尝试创建目录的次数
   var attempts = 0
   //最大尝试次数,默认为 10 次
   val maxAttempts = MAX_DIR_CREATION_ATTEMPTS
   var dir:File = null
   //只要 dir 为 null,一直循环尝试创建目录,直到最大尝试次数
   while(dir == null) {
       //尝试次数加 1
       attempts += 1
```

```
        //尝试创建目录的次数大于最大尝试次数,抛出 IOExcetion 异常
        if(attempts > maxAttempts){
          throw newIOException("Failed to create a temp directory(under" + root + ")after" +
            maxAttempts + " attempts!")
        }
        //尝试在给定的根目录下创建以 namePrefix 为前缀,以 UUID 产生的 UUID 字符串结尾的目录
        try {
          //得到目录对应的 File 对象
          dir = new File(root,namePrefix + " - " + UUID. randomUUID. toString)
          //若 dir 已经存在,或者不能创建该目录,置 dir 为 null
          if(dir. exists()||! dir. mkdirs()){
            dir = null
          }
        } catch { case e:SecurityException => dir = null; }//若没有权限创建目录,捕获 SecurityException 异常消息
      }
      //返回标准的创建文件夹的路径
      dir. getCanonicalFile
    }
```

上面源代码中有一个 attempts 变量，用于记录尝试创建文件夹的次数，如果 dir 文件夹为空，将会一直循环处理，直到 attempts 达到最大尝试次数为止。如果超过最大创建文件夹的次数，将会抛出 IOExeception 异常。创建文件夹将会以传入的 rootDir 为根路径，创建文件夹的格式为 prefix + " - " + UUID. randomUUID. toString 的形式，如果 dir 存在或者不能够创建目录，则将 dir 置为 null。若没有创建目录的权限，将捕获 SecurityException 异常，异常处理也是将 dir 置空，重新进行 while 循环。

在 DiskBlock 存储中，逻辑意义上的 block 和文件有什么关系呢？在 DiskBlock 中，每一个 Block 都被存储为一个文件，文件的名称是通过计算 BlockId 对象中的 fileName 的哈希值得到的，通过映射得到文件名称之后，将 block 中的数据写入到文件中。BlockId 对象中的 fileName 与文件路径映射关系的源代码如下所示。

```
def getFile(filename:String):File = {
  //由 fileName 计算出 hash 值
  val hash = Utils. nonNegativeHash(filename)
  //使用非负哈希值余上 localDirs 的长度,得到目录的 id
  val dirId = hash % localDirs. length
  //使用非负哈希值和 localDirs 做商,结果再与 subDirsPerLocalDir 做余运算,求得子目录 id
  val subDirId = (hash / localDirs. length)% subDirsPerLocalDir
 // 如果子目录不存在,则创建子目录
  val subDir = subDirs(dirId). synchronized {
      val old = subDirs(dirId)(subDirId)
```

```
          if(old! = null){
            //如果存在,则直接返回 subDir
            old
          } else {
            //如果不存在,则创建 subDir
            val newDir = new File(localDirs(dirId),"%02x".format(subDirId))
            //若 newDir 不存在或者不能创建目录,则抛出 IOException 异常
            if(! newDir.exists()&&! newDir.mkdir()){
              throw new IOException(s"Failed to create local dir in $ newDir.")
            }
            //将二维数组 subDirs 的 dirId 行、subDirId 的值设置成 newDir
            subDirs(dirId)(subDirId) = newDir
            //返回 newDir
            newDir
          }
      }
    //返回以 subDir 为根目录,filename 作为文件名称的 File 对象
    new File(subDir,filename)
  }
```

getFile 方法首先得到 fileName 的哈希值，用哈希值对 localDirs 数组长度做模运算，得到要存放数据的目录 ID，用 hash 值对 localDirs 数组长度做除运算，结果对 subDirsPerLocalDir 做模运算得到存放数据的目录中子目录的 ID。其中，subDirsPerLocalDir 的个数是通过 spark.diskStore.subDirectories 配置项配置的，默认值为 64。如果没有子目录，则创建子目录；如果有则直接返回，subdirs(i)有 sbudirs(i)锁保护，防止由于多线程的同时修改而造成数据的不一致。使用哈希映射将文件分配到不同的目录的目的是为了避免顶级目录的 inodes 过于庞大。

DiskStore 实现了 BlockManager 中存取 block 的方法，这里以 BlockStore 抽象类中的 putBytes(blockId:BlockId,_bytes:ByteBuffer,level:StorageLevel)为例，DiskStore 重写的 putBytes 方法的源代码如下。

```
override def putBytes(blockId:BlockId,_bytes:ByteBuffer,level:StorageLevel):PutResult = {
    //得到 bytes 的副本
    val bytes = _bytes.duplicate()
    logDebug(s"Attempting to put block $ blockId")
    //开始时间
    val startTime = System.currentTimeMillis
    //调用 getFile 方法,通过哈希映射得到对应的文件,这里的 blockId 是通过 CacheManager
  的 getOrCompute 方法得到的,getOrCompute 方法中有这样的代码:val key = RDDBlockId
  (rdd.id,partition.index),这里的 key 为通过 RDD 的 id 和 RDD 分区的索引得到的 BlockId,将
  RDD 中的 partition 转化为 Storage 模块中的 Block,具体转换可以查看 BlockId 类中的 RDD-
  BlockId 用例类
```




```
      val file = diskManager.getFile(blockId)
      //使用 nio,得到文件上的 channel 通道
       val channel = new FileOutputStream(file).getChannel
       Utils.tryWithSafeFinally {
         while(bytes.remaining > 0){
           //将 bytes 数据写入通道
           channel.write(bytes)
         }
       } {
         //关闭通道
         channel.close()
       }
       //得到完成时间
       val finishTime = System.currentTimeMillis
       logDebug("Block %s stored as %s file on disk in %d ms".format(
         file.getName,Utils.bytesToString(bytes.limit),finishTime - startTime))
       //返回一个 PutResult 对象,该对象携带以下信息。
       //1. 输入数据的估计大小
       //2. 返回放入的数据
       //3. 因此次 Put 数据而不得不 drop 掉的 block 列表,在 DiskStore 中,永远为空
       PutResult(bytes.limit(),Right(bytes.duplicate()))
   }
```

上面源代码中，putBytes 方法传入 blockId、_bytes 和 level 共 3 个参数，blockId 代表 BlockId 对象，该对象中有一个全局唯一的 name 属性；_bytes 是 ByteBuffer 类型，里面存放的是字节数据；level 表示 StorageLevel，代表存储级别。

putBytes 方法中通过调用 diskManager 的 getFile（blockId）方法，得到 BlockId 对象中 fileName 映射的 file 路径，然后通过该路径得到 File 对象，再使用 File 对象上的输出流通道向文件中写入字节数据，最后返回 PutResult 对象，它是一个携带此次 put 操作数据信息的对象。

Block 数据存入 DiskStore 之后，要怎么读取这些宝贵的数据呢？其实想要读取 DiskStore 中的 block 数据很简单，只需通过 BlockId 对象的 fileName 属性获得对应的哈希映射文件，并从文件中读取数据即可。下面来看一下 DiskStore 的 getBytes(blockId:BlockId,offSet:Long,length:Long)方法，源代码如下。

```
override def getBytes(blockId:BlockId):Option[ByteBuffer] = {
     //通过 diskManager 的 getFile 方法,通过计算 blockId 中 name 属性的哈希值匹配到文件
     val file = diskManager.getFile(blockId.name)
        //将 blockId 对应的文件传入 getByte(file,0,file.length)方法中,该方法放回 Option
   [ByteBuffer]
     getBytes(file,0,file.length)
```



```
}
private def getBytes(file:File,offset:Long,length:Long):Option[ByteBuffer] = {
  //以只读的形式获得 file 上的 channel 通道
val channel = new RandomAccessFile(file,"r").getChannel
    Utils.tryWithSafeFinally {
      //对于小文件,直接读取,不适用内存映射
      //文件长度小于 minMemoryMapBytes,直接读取。minMemoryMapBytes 的默认大小为 2M
      if(length < minMemoryMapBytes) {
        //使用 ByteBuffer 的 allocate 方法,创建长度为 length 的字节缓存数组
        val buf = ByteBuffer.allocate(length.toInt)
        //设置读写文件的起始位置为 offset 处,此处为 0,表示文件开始
        channel.position(offset)
        //当 buf 中还有字节
        while(buf.remaining()! =0) {
          //从 channel 中顺序读取一个字节放入 buf,若读到 -1,抛出 IOException 异常
          if(channel.read(buf) == -1) {
            throw newIOException("Reached EOF before filling buffer\n" +
                s"offset = $ offset\nfile = $ {file.getAbsolutePath}\nbuf.remaining = $ {buf.remaining}")
          }
        }
        //翻转数组



        buf.flip()
        //返回字节缓存数组
        Some(buf)
      } else {
        //通过 channel 上的 map 方法,以内存映射的方式读取文件内容
        Some(channel.map(MapMode.READ_ONLY,offset,length))
      }
    } {
      //关闭通道
      channel.close()
    }
}
```

getBytes(blockId:BlockId)方法中，通过 BlockId 对象中的 name 属性，由哈希算法获取映射的文件，并通过 getBytes(file:File,offSet:Long,length:Long)读取 Block 映射文件中的字节数据。对于小于 2 MB 的文件，采用直接读取的方式；对于大于 2 MB 的文件，采用内存映

射的方式读取。内存映射方式可以大大提高读取数据的性能，尤其适合读取大文件，这里的minMemoryMapBytes 取值默认为 2 MB。

从 getBytes 和 putBytes 两个方法的分析中可以看到，在 DiskStore 中读取和保存 block 都是先通过 BlockId 获得映射的文件，然后通过数据流的形式读取和保存文件的。

DiskStore 会将数据保存到磁盘，如果频繁地对磁盘进行 I/O 操作，必将严重影响系统的性能，正好 MemoryStore 可以解决对磁盘的 I/O 操作带来的问题，由于 MemoryStore 数据是驻留在内存中的，可以直接从内存中读取数据，减少磁盘 I/O，大大加快了数据的读写速度，因此 MemoryStore 可以给系统性能带来巨大的提升。下面就来看看 MemoryStore 的源代码实现，MemoryStore 与 DiskStore 最大的不同是 MemoryStore 将 block 保存在内存中，而 DiskStore 将 block 通过文件的形式保存在磁盘中。

2. Memory Store

MemoryStore 中维护着一个 LinkedHashMap 来管理所有的 Block，Block 被包装成 MemoryEntry 对象，该对象中保存了 Block 相关的数据信息，如大小、是否反序列化等。LinkedHashMap 以 blockId 作为键，blockId 对应的 MemoryStore 作为值，其源代码的定义如下。

```
//链表类型的哈希 map,它保留了数据插入的顺序,构造 LinkedHashMap 时传入 3 个参数,第一个参数表示初始化容量,这里取值为 32,第二个参数表示负荷系数,这里取值 0.75,容量超过 32 ×0.75 =24 个时,自动扩容。第三个参数表示访问模型,true 表示访问顺序,false 表示插入顺序
private val entries = new LinkedHashMap[BlockId,MemoryEntry](32,0.75f,true)
//MemoryEntry 用例类
private case class MemoryEntry(value:Any,size:Long,deserialized:Boolean)
```

MemoryStore 中必须要有足够的内存来存放 Block，否则将会把 Block 溢出到磁盘文件中。下面以 putBytes 方法为例，源代码如下。

```
override def putBytes(blockId:BlockId,_bytes:ByteBuffer,level:StorageLevel):PutResult = {
    // Work on a duplicate - since the original input might be used elsewhere.
    //得到_bytes 的副本
    val bytes = _bytes.duplicate()
    // Rewinds this buffer.  The position is set to zero and the mark is discarded.
    bytes.rewind()
    //判断是否反序列化
    if(level.deserialized){
      //反序列化字节数组为可迭代的 value
       val values = blockManager.dataDeserialize(blockId,bytes)
       //调用 putIterator 方法
       putIterator(blockId,values,level,returnValues = true)
     } else {
       //需要溢出的 block 数组,数组中存放的是(BlockId,BlockStatus)二维元组
       val droppedBlocks = new ArrayBuffer[(BlockId,BlockStatus)]
```

```
        //调用 tryToPut 方法,尝试 put 操作,若内存中不能存放,则溢出到磁盘
        tryToPut(blockId,bytes,bytes.limit,deserialized = false,droppedBlocks)
        PutResult(bytes.limit(),Right(bytes.duplicate()),droppedBlocks)
    }
}
```

putBytes 方法中包含 BlockId、ByteBuffer、StorageLevel 共 3 个参数，这 3 个参数的含义与 DiskStore 中 putBytes 参数一样。代码行中，通过 duplicate 方法得到_bytes 的副本，以避免在多个地方修改一个变量，造成逻辑上的错误。接下来判断 StorageLevel 是否为 deserialized，如果为 true，则进行反序列化，将字节数组反序列化成可迭代的对象，然后调用 putIterator 方法存入内存；若 deserialized 为 false，则调用 tryToPut 方法尝试将字节数组中的数据存入内存，tryToPut 方法的源代码如下。

```
private def tryToPut(
      blockId:BlockId,
      value:() => Any,
      size:Long,
      deserialized:Boolean,
      droppedBlocks:mutable.Buffer[(BlockId,BlockStatus)]):Boolean = {
  //使用 memoryManager 对象上的 synchronized 同步块,同步操作,防止多线程访问
  memoryManager.synchronized {
    //为任务释放内存
    releasePendingUnrollMemoryForThisTask()
    //请求足够的内存,用于存放 block 数据
    val enoughMemory = memoryManager.acquireStorageMemory(blockId,size,droppedBlocks)
   //如果 enoughMemory 为 true
    if(enoughMemory){
      //有足够的内存,用 block 数据构建 MemoryEntry 对象
      val entry = new MemoryEntry(value(),size,deserialized)
      entries.synchronized {
        //entries 是一个 LinkedHashMap,向该 map 中存入 block 数据信息
        entries.put(blockId,entry)
      }
    } else {//如果内存不足以存放下 block 信息,则将其溢出到 Disk Store,前提是允许使用 DiskStore
      lazy val data = if(deserialized){
        Left(value().asInstanceOf[Array[Any]])//若 deserialized 为 true,将 values 转换成 Array 数组
      } else {
        Right(value().asInstanceOf[ByteBuffer].duplicate())//否则,将 value 作为 ByteBuffer 实例
      }
```

```
          //调用 blockManager 的 dropFromMemory 方法,将 block 数据从内存溢出到磁盘
          val droppedBlockStatus = blockManager.dropFromMemory(blockId,() => data)
          droppedBlockStatus.foreach { status => droppedBlocks += ((blockId,status))}
        }
      //返回内存是否满足存储的标志 enoughMemory
      enoughMemory
    }
  }
```

在 tryToPut 方法中，首先调用 MemoryManager 的 acquireStorageMemory 方法，判断空闲内存是否足以容纳 block。若可以容纳该 Block，则将该 Block 放入内存中的 entries 这个 LinkedHashMap 进行管理；若不足以容纳，则通过调用 dropFromMemory 方法将此 Block 溢出到 DiskStore。

MemoryStore 将 Block 放入 Executor 内存中，每一个独立的 JVM 进程都可以分配堆大小（-Xmx 和 -Xms）。Executor 详细的内存模型分析在 6.4 小节介绍。

接下来讲解如何从 MemoryStore 中读取 Block 信息。从 MemoryStore 中取得 Block 比较简单，根据 blockId 从哈希表中取出即可。这里以 MemoryStore 的 getValues 方法为例，源代码如下。

```
override def getValues(blockId:BlockId):Option[Iterator[Any]] = {
  //通过 entyies 这个 LinkedHashMap 的 synchronized 同步块
    val entry = entries.synchronized {
      //取出 blcokId 对应的 MemoryEntry 信息
      entries.get(blockId)
    }
    if(entry == null){
      //若 entry 为 null ,返回 None
      None
    } else if(entry.deserialized){
      //如果 entry 的 deserialized 为 true,将 value 转换成 Array 数组,返回数组的迭代器
      Some(entry.value.asInstanceOf[Array[Any]].iterator)
    } else {
      //得到 value 的一个只读副本,类型为 ByteBuffer
      val buffer = entry.value.asInstanceOf[ByteBuffer].duplicate()// Doesn't actually copy data
      Some(blockManager.dataDeserialize(blockId,buffer))
    }
  }
```

getVaults 方法接受 blockId 作为参数，从 entries 这个 LinkedHashMap 中查找，如果没有查到该 blockId 对应的 MemoryEntry，返回 None，否则再根据是否反序列化得到反序列化之后的数据并返回。

Spark 中有 3 种 BlockStore 的实现，分别是 DiskStore、MemoryStore 和 ExternalBlockStore，上面谈到了 DiskStore 和 MemoryStore 中存取 block 的操作，接下来看一下 ExternalBlockStore 中对 block 的存取操作。

3. External BlookStore

ExternalBlockStore 将 Block 存储在 JVM 外部的存储系统中，Spark 目前实现的 ExternalBlockStore 将 block 存放在 Tachyon 分布式内存文件系统中。ExternalBlockStore 的 StorageLevel 是 OFF_HEAP。

ExternalBlockStore 中是如何存取 block 的呢？仍然以 putBytes 方法为例，ExternalBlockStore 中 putBytes 方法的源代码如下所示。

```
override def putBytes(blockId:BlockId,bytes:ByteBuffer,level:StorageLevel):PutResult = {
    putIntoExternalBlockStore(blockId,bytes,returnValues = true)
  }
```

putBytes 方法接受 3 个参数，分别是 BlockId、ByteBuffer 和 StorageLevel，这 3 个参数的含义同 DiskStore 和 MemoryStore 的 putBytes 方法的参数，分别代表 Block 对象，Block 对应的字节数组和存储级别，这里 StorageLevel 的取值为 OFF_HEAP，该方法返回 PutResult 对象。在 putBytes 方法内部调用 putIntoExternalBlockStore 方法，该方法用于将 Block 消息存入 ExeternalBlockStore 中，其源代码如下所示。

```
private def putIntoExternalBlockStore(
      blockId:BlockId,
      bytes:ByteBuffer,
      returnValues:Boolean):PutResult = {
    logTrace(s"Attempting to put block $ blockId into ExternalBlockStore")
    // we should never hit here if externalBlockManager is None. Handle it anyway for safety.
    try {
     //开始时间
  val startTime = System.currentTimeMillis
      //externalBlockManager 是否被定义
      if(externalBlockManager.isDefined){
        //调用 duplicate 方法得到 bytes 的只读副本
        val byteBuffer = bytes.duplicate()
        byteBuffer.rewind()
        //调用 externalBlockManager 的 putBytes 方法,将 byteBuffer 存入 ExternalBlockStore 中
        externalBlockManager.get.putBytes(blockId,byteBuffer)
        //得到 byteBuffer 的大小信息
        val size = bytes.limit()
        //如果 returnValues 为 true,返回 Right(bytes),否则返回 null
        val data = if(returnValues){
          Right(bytes)
        } else {
```

```
            null
          }
          //完成时间
          val finishTime = System. currentTimeMillis
          //返回 PutResult 对象
          PutResult(size,data)
        } else {
          //如果没有定义 externalBlockManager,打印错误日志
            logError(s" Error in putBytes( $ blockId):no ExternalBlockManager has been configured")
          PutResult( -1,null,Seq((blockId,BlockStatus. empty)))
        }
    }  }
```

将 Block 存放到 JVM 外部存储系统中，必须借助 ExternalBlockManager，首先判断 ExternalBlockManger 是否定义，如果没有定义，则打印出 ExternalBlockManger 没有定义的信息；如果定义了 ExternalBlockManger，调用 externalBlockManger. get. putBytes 方法将 byteBuffer 存入外部系统中。

这里提到了 ExternalBlockManager，它是一个抽象类，定义了使用外部存储系统存取 Block 的方法。ExternalBlockManager 的源代码如下所示。

```
private[spark] abstract class ExternalBlockManager {
//定义的 BlockManager 对象,因为 BlockManager 是和其他模块交互的类,其中有一些有用的方法,可以在 ExternalBlockManager 中使用,如 dataDeserialize、blockManager. conf 对象等
 protected var blockManager:BlockManager = _
//重写的 toString 方法
 override def toString:String = {"External Block Store"}
//初始化方法,该方法传入 BlockManager 和 Executor 的 id 编号
 def init(blockManager:BlockManager,executorId:String):Unit = {
   this. blockManager = blockManager
}
 //移除 blockId 对应的 block 数据
  def removeBlock(blockId:BlockId):Boolean
 //检查 blockId 对应的 block 是否存在
  def blockExists(blockId:BlockId):Boolean
 //加入 block,值为 ByteBuffer 类型
  def putBytes(blockId:BlockId,bytes:ByteBuffer):Unit
 //加入 block,值为可迭代类型
  def putValues(blockId:BlockId,values:Iterator[_]):Unit = {
    //调用 blockManager 的 dataSerialize 方法,将可迭代的 values 数据序列化成 ByteBuffer 数据
    val bytes = blockManager. dataSerialize(blockId,values)
```

```
    //调用 putBytes 方法,加入 block 数据
      putBytes(blockId,bytes)
  }
  //查询并返回 blockId 对应的 block 数据,返回类型为 Option[ByteBuffer]
  def getBytes(blockId:BlockId):Option[ByteBuffer]
  //查询并返回 blockId 对应的 block 数据,返回类型为 Option[Iterator[_]]
  def getValues(blockId:BlockId):Option[Iterator[_]] = {
    getBytes(blockId).map(buffer => blockManager.dataDeserialize(blockId,buffer))
  }
  //得到 blockId 对应的 block 的大小
  def getSize(blockId:BlockId):Long
  //关闭 ExternalBlockManager,这是一个钩子方法
  def shutdown()
}
```

ExternalBlockManager 抽象类定义了使用外部系统存储 Block 的基本的抽象方法，由具体存储系统实现。

ExternalBlockManager 中的 init 方法用于初始化 ExternalBlockManager，该方法有两个参数，第一个参数是 BlockManager，第二个参数是 String 类型的 executorId，BlockManager 定义了供外部模块调用的公共接口，BlockManager 中提供了数据序列化和反序列化的方法，在 ExternalBlockManager 中可以使用以下几个方法。

- removeBlock，根据 blockId 删除对应的 Block。
- BlockExists，判断 blockId 对应的 Block 在外部存储系统中是否存在。
- putBytes，将 blockId 对应的数据存入外部存储系统。
- putValues，将 blockId 对应的数据存入外部存储系统。
- getBytes，通过 blockId，取出对应的 Block。
- getValues，通过 blockId，取出对应的 Block。
- getSize，返回 blockId 对应的 Block 的大小。

shutdown 在系统关闭时被调用，用于清理外部存储系统中持久化了的数据，例如，在 TachyonBlockManager 中，该方法用于递归删除某些保存数据的目录。

目前在 Spark 中，ExternalBlockManager 抽象类只有一个实现，即针对 Tachyon 的实现，这个实现类是 org. apache. spark. storage. TachyonBlockManager，接下来看一下 TachyonBlockManager 中重写的相关方法。

先来看一下 TachyonBlockManager 中 init 方法的实现，源代码如下。

```
override def init(blockManager:BlockManager,executorId:String):Unit = {
    super.init(blockManager,executorId)
    //得到 Tachyon 中设置的 baseDir,该值可以通过 spark.externalBlockStore.baseDir 配置
  val storeDir = blockManager.conf.get(ExternalBlockStore.BASE_DIR,"/tmp_spark_tachyon")
    //得到保存文件夹的名称,该值可以通过 spark.externalBlockStore.folderName 配置
val appFolderName = blockManager.conf.get(ExternalBlockStore.FOLD_NAME)
```

```
//由 storeDir、appFolderName 和 executorId 拼装成 rootDirs
rootDirs = s"$storeDir/$appFolderName/$executorId"
//获得 Tachyon 的 master 配置,默认为 tachyon://localhost:19998,该值可以通过 spark.externalBlockStore.url 进行配置
master = blockManager.conf.get(ExternalBlockStore.MASTER_URL,"tachyon://localhost:19998")
//如果 master 不为空,通过 TachyonFs 的 get 方法,由 master 地址得到 TachyonFS 对象
client = if(master != null && master != ""){
  TachyonFS.get(new TachyonURI(master),new TachyonConf())
} else {
  //若 master 为空,返回 null
  null
}
//若 client 为 null,打印错误日志并抛出 IOException 异常
if(client == null){
  logError("Failed to connect to the Tachyon as the master address is not configured")
  throw newIOException("Failed to connect to the Tachyon as the master" +
    "address is not configured")
}
//得到每一个 Tachyon 目录下子目录的个数,该个数可以通过 spark.external BlockStore.subDirectories 配置
subDirsPerTachyonDir = blockManager.conf.get("spark.externalBlockStore.subDirectories",
  ExternalBlockStore.SUB_DIRS_PER_DIR).toInt
//使用 rootDirs 创建根目录,并在根目录中创建子目录
tachyonDirs = createTachyonDirs()
//创建二维数组
subDirs = Array.fill(tachyonDirs.length)(new Array[TachyonFile](subDirsPerTachyonDir))
//为每一个 Tachyon 目录注册 shutdown 回调函数,当发生 shutdown 时,删除目录
tachyonDirs.foreach(tachyonDir => ShutdownHookManager.registerShutdownDeleteDir(tachyonDir))
}
```

init 方法中，通过传入的 BlockManager 中的 SparkConf 对象取出外部存储的路径，该路径通过 spark. externalBlockStore. baseDir 配置，并通过 SparkConf 对象取出 Tachyon 保存数据文件夹的名称，该名称可以通过 spark. externalBlockStore. folderName 配置。使用 storeDir、appFolderName 和 executorId 拼接成一个数据块存储的根路径。该路径格式为/storeDir/appFolderName/executorId。通过 SparkConf 对象取出 Tachyon 的配置 URL，该配置项可以通过 spark. externalBlockStore. url 来配置。得到 Tachyon 的 master 地址之后，通过 TachyonFS. get (new TachyonURI(master),new TachyonConf())方法，得到 Tachyon 分布式文件系统的引用 TachyonFS，通过该 client，可以非常方便地使用 TachyonFS 进行创建、删除和存取数据操作。

init 方法中另外一个重要的操作是创建目录，首先通过 spark. externalBlockStore. subDirectories 配置项获得子目录的个数，默认情况下为 64 个，可以通过 spark. externalBlockStore.

subDirectories 配置项来配置。第 28 行调用 createTachyonDirs 方法，使用拼凑出来的rootDirs创建目录。创建好目录后，使用 subDirectories 参数构建一个二维数组，该二维数组有 tachyonDirs. length 行，subDirsPerTachyonDir 列。

init 方法完成 Tachyon 目录的创建准备工作，那么 TachyonBlockManager 是如何存取数据的呢？还是以 putBytes 方法为例，putBytes 的源代码如下所示。

```
1. override def putBytes(blockId:BlockId,bytes:ByteBuffer):Unit = {
2.     //调用 getFile 方法,使用哈希函数将 blockId 映射到一个文件
3. val file = getFile(blockId)
4.     //得到文件上的输出流
5.     val os = file. getOutStream(WriteType. TRY_CACHE)
6.     try {
7.       //调用 FileOutStream 上的 write 方法写入字节数据
8.       os. write(bytes. array())
9.     } catch {
10.        //捕获异常,关闭流
11.        case NonFatal(e) =>
12.          logWarning(s"Failed to put bytes of block $ blockId into Tachyon",e)
13.          os. cancel()
14.      } finally {
15.        //关闭流
16.        os. close()
17.      }
18.   }
```

该方法传入两个参数，第一个参数是 BlockId 对象，该对象中包含 name 等属性及方法。第二个参数是 ByteBuffer 数组，为待存储的字节数据。上面源代码的第 3 行调用 getFile 方法，得到 blockId 经哈希映射后的文件，得到文件对象，使用文件上的输出流将字节数据写入文件中。

通过 TachyonBlockManager 查找 Block 信息也非常简单，只需传入 blockId，在 Tachyon 系统中找出对应的数据返回即可。这里以 getBytes 方法为例，源代码如下。

```
override def getBytes(blockId:BlockId):Option[ByteBuffer] = {
    //调用 getFile 方法,使用 blockId 中的 name 属性,由哈希函数映射得到文件
    val file = getFile(blockId)
    //如果 file 为空或者 TachyonFile 的 locationHosts 大小为 0,则返回 None
    if(file == null || file. getLocationHosts. size == 0){
      return None
    }
    //得到 TachyonFile 上的输入流
    val is = file. getInStream(ReadType. CACHE)
     try {
```

```
        //文件大小
        val size = file.length
        //创建 length 长度的字节数组
        val bs = new Array[Byte](size.asInstanceOf[Int])
        //使用 ByteStreams 的 readFully 方法从输入流读取数据到 bs 中
        ByteStreams.readFully(is,bs)
        //返回读取到的数据
        Some(ByteBuffer.wrap(bs))
      } catch {
        //捕获异常,返回 None
        case NonFatal(e) =>
          logWarning(s"Failed to get bytes of block $blockId from Tachyon",e)
          None
      } finally {
        //关闭输入流
        is.close()
      }
    }
```

在 getBytes 方法中，调用 getFile 方法，找出 blockId 经哈希函数映射后对应的文件，通过文件的输入流，读出字节数据并返回。

至此，DiskStore、MemoryStore 和 ExternalBlockStore 都已讲解完。在 Spark 中拥有不同的 BlockStore 实现，并且这些实现有可能继续增加，为了对用户屏蔽这些内部的细节，并保证 Spark 今后对不同存储的扩展，Spark Storage 模块对外提供了一个统一的方法类 BlockManager。在 BlockManager 中提供了存取数据的方法，不必关心不同 Store 的具体实现细节，只需通过配置使用不同的 StorageLevel，BlockManager 就能自动选择不同的存储，并调用该存储的方法。

4. 通过 BlockManager 读写数据

既然 BlockManager 是一个统一的接口，那么在 BlockManager 中肯定提供了读写数据的方法，并且在这些方法中会根据不同的 StorageLevel，调用 BlockStore 接口实现类的方法来读写数据。接下来看一下 BlockManager 提供的读写方法，BlockManager 提供了 putBlockData 方法，用于存储数据，该方法的源代码如下。

```
override def putBlockData(blockId:BlockId,data:ManagedBuffer,level:StorageLevel):Unit = {
  //调用 putBytes 方法存放数据
  putBytes(blockId,data.nioByteBuffer(),level)
}
```

该方法有 3 个参数，第一个参数是 BlockId 对象；第二个参数是 ManagedBuffer，表示待存数据；第三个参数是 StorageLevel，用于指定存储的级别。方法中调用 putBytes 方法，该方法的源代码如下所示。



```
def putBytes(
      blockId:BlockId,
      bytes:ByteBuffer,
      level:StorageLevel,
      tellMaster:Boolean = true,
      effectiveStorageLevel:Option[StorageLevel] = None):Seq[(BlockId,BlockStatus)] = {
    //要求 bytes 不为空
    require(bytes! = null,"Bytes is null")
    //调用 doPut 方法,存入数据
    doPut(blockId,ByteBufferValues(bytes),level,tellMaster,effectiveStorageLevel)
}
```

在 putBytes 方法中，tellMaster 参数用于设置在存入数据后是否通知 master，默认为 true。还有一个参数是 effectiveStorageLevel，设置有效的存储级别，默认为 None。require 用于检测传入的 bytes 是否为 null，如果为空将以“Bytes is null”作为异常消息并抛出 IllegalArgumentException 异常。putBytes 方法主要完成将一个序列化的字节数组存入 BlockManager，并返回一个更新后的块列表。该方法中调用了 doPut 方法，源代码如下。

```
private def doPut(
      blockId:BlockId,
      data:BlockValues,
      level:StorageLevel,
      tellMaster:Boolean = true,
      effectiveStorageLevel:Option[StorageLevel] = None)
    :Seq[(BlockId,BlockStatus)] = {
//检查 blockId 是否为 null,若为 null,抛出 IllegalArgumentException 异常
    require(blockId! = null,"BlockId is null")
    //level 不能为 null,并且 level 合法,否则抛出 IllegalArgumentException 异常
require(level! = null && level.isValid,"StorageLevel is null or invalid")
    //检查 effectiveStorageLevel
    effectiveStorageLevel.foreach { level =>
      require(level! = null && level.isValid,"Effective StorageLevel is null or invalid")
    }
    //新建数组,该数组用于返回数据。数组中存放用 BlockId 和 BlockStatus 组成的二维
  元组
    val updatedBlocks = new ArrayBuffer[(BlockId,BlockStatus)]
    //put 操作产生的 Block 信息
    val putBlockInfo = {
      //构建 BlockInfo 对象 tinfo
      val tinfo = new BlockInfo(level,tellMaster)
      //如果 blockInfo 这个 TimeStampedHashmap 中没有 blockId,则加入此 BlockId 和 tinfo,
  putIfAbsent 方法返回 key 对应的更新此 key 的 value 之前的 value,即保留老的信息
```




```
      val oldBlockOpt = blockInfo. putIfAbsent(blockId,tinfo)
      //如果 oldBlockOpt 有定义,说明 blockInfo 这个 TimeStampedHashmap 中有 blockId 对应的信息,直接返回 updatedBlocks,此时 updatedBlocks 为一个 ArrayBuffer 对象,对象内无值
      if(oldBlockOpt. isDefined) {
        if(oldBlockOpt. get. waitForReady()) {
          logWarning(s"Block $blockId already exists on this machine; not re-adding it")
          return updatedBlocks
        }
        oldBlockOpt. get
      } else {
        //若 oldBlockOpt 没有定义,返回 tinfo 对象
        tinfo
      }
    }
    //开始时间
    val startTimeMs = System. currentTimeMillis
    var valuesAfterPut:Iterator[Any] = null
    var bytesAfterPut:ByteBuffer = null
    var size = 0L
//进行 put 操作的有效的 StorageLevel
    val putLevel = effectiveStorageLevel. getOrElse(level)
    //启动一个线程,在本地存储之前,异步初始化好要进行冗余备份的数据。这有助于提高发送数据的速度,从而提高性能
    val replicationFuture = data match {
      case b:ByteBufferValues if putLevel. replication > 1 =>
        //如果 replication 设置为大于 1,则通过 ByteBuffer 的 duplicate 方法得到一个只读的副本数据
        val bufferView = b. buffer. duplicate()
        Future {
          //在线程池中执行 replicate 操作,这是一个阻塞型操作,该操作将会把 blockId 对应的 bufferView 数据复制到另一个结点上,这是一个阻塞型操作,待复制完成后才返回
          replicate(blockId,bufferView,putLevel)
        }(futureExecutionContext)
      case _ => null
    }
  //防止其他线程 get 这个 block,使用该 putBlockInfo 上的同步块进行同步操作,直到 marked 成为 true
  putBlockInfo. synchronized {
  //marked 此时为 false
    var marked = false
      try {
        //returenValues 表示是否返回 put 操作的值,blockStore 表示存储方式
```



```
        val(returnValues,blockStore:BlockStore) = {
          //内存存储,返回 true 和 memoryStore
          if(putLevel.useMemory){
            //首先将数据放入内存,若在 useDisk 为 true 的情况下内存不能满足存储,将会把此 block 溢出到磁盘
            (true,memoryStore)
          }
          //OFF_HEAP 存储,返回 false 和 externalBlockStore
  else if(putLevel.useOffHeap){
            (false,externalBlockStore)
          }
          //磁盘存储,diskStore
  else if(putLevel.useDisk){
            // Don't get back the bytes from put unless we replicate them
            (putLevel.replication > 1,diskStore)
          } else {
            //若没有设置 StoreageLevel,抛出 BlockException 异常
            assert(putLevel == StorageLevel.NONE)
            throw newBlockException(
              blockId,s"Attempted to put block $blockId without specifying storage level!")
          }
        }
    // 匹配 put 操作存入的值的类型,调用 blockStore 的方法
        val result = data match {
          //可迭代值类型,调用 blockStore 的 putIterator 方法
          case IteratorValues(iterator) =>
            blockStore.putIterator(blockId,iterator,putLevel,returnValues)
          //数组类型的值,调用 blockStore 的 putArray 方法
          case ArrayValues(array) =>
            blockStore.putArray(blockId,array,putLevel,returnValues)
          //ByteBufferValues 类型的值,调用 blcokStore 的 putBytes 方法
          case ByteBufferValues(bytes) =>
            bytes.rewind()
            blockStore.putBytes(blockId,bytes,putLevel)
        }
        //获得结果大小
        size = result.size
        //对结果数据类型进行匹配,返回 put 操作后的 result
        result.data match {
          case Left(newIterator) if putLevel.useMemory => valuesAfterPut = newIterator
          case Right(newBytes) => bytesAfterPut = newBytes
           case _ =>
```

```
                }
                //如果使用内存存储,遍历 result 结果中的 droppedBlocks,将溢出到磁盘的 block 添加到 updatedBlocks 数组中
                if(putLevel.useMemory){
                    result.droppedBlocks.foreach { updatedBlocks += _ }
                }
                //得到当前 block 的状态
                val putBlockStatus = getCurrentBlockStatus(blockId,putBlockInfo)
                //判断状态中的 storageLevel 不为 NONE
                if(putBlockStatus.storageLevel != StorageLevel.NONE){
                    //将 marked 标志设置成 true
                    marked = true
                    //调用 putBlockInfo 的 markReady 方法,标记该 block 写操作完成
                    putBlockInfo.markReady(size)
                    //如果 tellMaster 为 true,调用 reportBlockStatus 方法,报告状态
                    if(tellMaster){
                        reportBlockStatus(blockId,putBlockInfo,putBlockStatus)
                    }
                    updatedBlocks += ((blockId,putBlockStatus))
                }
            }
        }
        //如果 replication 大于 1,异步开始数据的复制操作
        if(putLevel.replication > 1){
            data match {
                case ByteBufferValues(bytes) =>
                    if(replicationFuture != null){
                        //启动 replicationFuture 线程
                        Await.ready(replicationFuture,Duration.Inf)
                    }
                    replicate(blockId,bytesAfterPut,putLevel)
            }
        }

    BlockManager.dispose(bytesAfterPut)
    //返回 updatedBlocks
    updatedBlocks
    }
```

BlockManager 中的 doPut 是一个很复杂的方法，下面将逐一讲解该方法中的代码。该方法根据 StorageLevel 设置的值将 block 存入对应的 BlockStore 中，如果有必要将会产生并保存副本。

在该方法中，首先检查 blockId、storageLevel 和 effectiveStorageLevel 是否为空，如果为空将会抛出 IllegalArgumentsException 异常并返回。新建一个名为 updatedBlocks 的 ArrayBuffer，该缓存数组中存放一个二维元组，元组第一位存放 BlockId 对象，第二位存放 BlockStatus 对象，该 updatedBlocks 数组用于返回更新后的 blocks 信息。putBlockInfo 代码块中构建 BlockInfo 对象，首先会到 blockinfo 这个 TimeStampedHashmap 类型的 map 中查找 blockId 对应的 BlockInfo 是否已经定义，如果没有查到对应的 BlockInfo 对象，使用 storageLevel 和 tellMaster 构建一个 BlockInfo，并存入 blockInfo 中，返回构建好的 BlockInfo；如果已经定义了该 BlockInfo，返回旧的 BlockInfo。

如果 StorageLevel 中配置的 replication 数目大于 1，这种情况下，将在 futureExecutionContext 线程池中启动线程，在线程中进行数据的备份操作。

在 putBlockInfo 同步代码块中，根据不同的 StorageLevel，判断使用不同的 BlockStore，并判断是否需要返回值。得到 blockStore 之后，判断 data 的组织方式，调用 blockStore 的具体存储方法实际存储 Block 数据。

如果 tellMaster 为 true，将调用 reportBlockStatus(blockId,putBlockInfo,putBlockStatus)方法，向 master 报告 Block 的状态。最后根据 StorageLevel 中的 replication 数目产生副本。

在 doPut 函数中，只需完成以下几项工作。

1）为 Block 创建 BlockInfo，存储 block 的相关信息。

2）为 blockInfo 加锁，使其他线程不能访问。

3）根据 StorageLevel，将 Block 存到 Memory 或 Disk 或 ExternalBlockStore 中。

4）解锁 blockInfo，使其他线程可访问。

5）根据 StorageLevel 中的 replication 判断是否将 Block 制作成副本存放。

接下来了解一下在 BlockManager 中是如何获取数据的，首先来看一下 get 方法，其源代码如下所示。

```
//该方法用于在本地结点或者远程结点上返回 blockId 对应的 block 数据
def get(blockId:BlockId):Option[BlockResult] = {
    //首先调用 getLocal 方法,查找本地是否存在 blockId 对应的 block 数据
    val local = getLocal(blockId)
    //如果有定义,方法直接返回 local 数据
    if(local.isDefined){
      logInfo(s"Found block $blockId locally")
      return local
    }
    //在本地没有定义该 blcokId 的情况下,查询远程结点上是否存在该 blockId 对应的 block 数据
    val remote = getRemote(blockId)
    //若远程结点上有定义该 blockId,直接返回 remote
    if(remote.isDefined){
      logInfo(s"Found block $blockId remotely")
      return remote
```

```
    }
    //如果本地和远程结点上都没有该 blockId 对应的 blcok,返回 None
    None
}
```

get 方法中，根据 blockId，首先调用 getLocal 方法，从本地 blockManager 中获得数据。如果获得了数据则返回；如果没有获取到数据，调用 getRemote 方法，尝试在其他远程 blockManager 中查找数据，如果找到则返回，没有找到则返回 None。Spark 中，任务往往是根据 block 所在位置进行分配的，大部分情况下通过 getLocal 就能找到 blockId 对应的数据，但是在资源有限的情况下，任务调度器可能将任务调度到与 Block 所在结点不同的结点上执行，这种情况下必须通过 getRemote 方法得到远程 Block 数据。先来看一下 getLocal 方法，如下所示。

```
def getLocal(blockId:BlockId):Option[BlockResult] = {
    logDebug(s"Getting local block $blockId")
    //调用 doGetLocal 方法
    doGetLocal(blockId,asBlockResult = true).asInstanceOf[Option[BlockResult]]
}
```

getLocal 方法根据传入的 BlockId 通过调用 doGetLocal 方法取得 BlockId 对应的 Block。doGetLocal方法的源代码如下所示。

```
private def doGetLocal(blockId:BlockId,asBlockResult:Boolean):Option[Any] = {
    //首先从 blockInfo 这个 TimeStampedHashmap 中查找是否有 blockId
    val info = blockInfo.get(blockId).orNull
    //如果有,说明本地查找 blockId 成功,返回 blockId 对应的数据
    if(info != null){
      //避免多线程访问该 block,使用 info 的 synchronized 同步块
      info.synchronized {
        //再次检查 blockInfo 中是否存在 blockId,因为 removeBlock 方法可能会将该 blockId 对应的数据移除,这里是双保险
        if(blockInfo.get(blockId).isEmpty){
          logWarning(s"Block $blockId had been removed")
          //如果再次检查发现没有了 blockId 对应的数据,返回 None
          return None
        }

        //如果在 info 这个 block 上有其他线程正在进行写操作,则等待,直到其他线程操作完成
        if(!info.waitForReady()){
          //如果等待失败,返回 None
          logWarning(s"Block $blockId was marked as failure.")
          return None
        }
```



```
        val level = info.level
        //若使用的是 MEMORY 存储级别,查找 MemoryStore 并返回数据
        if(level.useMemory){
          logDebug(s"Getting block $blockId from memory")
          val result = if(asBlockResult){
                    memoryStore.getValues(blockId).map(new BlockResult(_, DataReadMethod.Memory,info.size))
          } else {
            memoryStore.getBytes(blockId)
          }
          result match {
            case Some(values) =>
              return result
            case None =>
              logDebug(s"Block $blockId not found in memory")
          }
        }
        //如果使用的是 OFF_HEAP 存储级别,查找 ExternalBlockStore 返回数据
        if(level.useOffHeap){
          logDebug(s"Getting block $blockId from ExternalBlockStore")
          if(externalBlockStore.contains(blockId)){
            val result = if(asBlockResult){
              externalBlockStore.getValues(blockId)
                .map(newBlockResult(_,DataReadMethod.Memory,info.size))
            } else {
              externalBlockStore.getBytes(blockId)
            }
            result match {
              case Some(values) =>
                return result
              case None =>
                logDebug(s"Block $blockId not found in ExternalBlockStore")
            }
          }
        }
        //如果使用的是 DISK 存储级别,通过 DiskStore 查找并返回数据
        if(level.useDisk){
          logDebug(s"Getting block $blockId from disk")
          val bytes:ByteBuffer = diskStore.getBytes(blockId) match {
            case Some(b) => b
            case None =>
              throw newBlockException(
```

```
                blockId,s"Block $ blockId not found on disk,though it should be")
          }
          assert(0 == bytes. position())
          //判断是否启用内存存储
          if(! level. useMemory){
            //若不能存储在内存中,直接返回数据即可
            if(asBlockResult){
                  return Some(newBlockResult(dataDeserialize(blockId, bytes), DataReadMethod. Disk,
                info. size))
            } else {
              return Some(bytes)
            }
          } else {
            //若可以存入内存中,将查出的数据放入内存,这样下次再查找该 block 数据时,直接从内存中获取即,以提高速度
            if(! level. deserialized ||! asBlockResult){
              memoryStore. putBytes(blockId,bytes. limit,() => {
                val copyForMemory = ByteBuffer. allocate(bytes. limit)
                //如果内存不能放下该 block 数据,将会发生 OOM,因此将该操作放入`() => ByteBuffer`中并且懒加载
                copyForMemory. put(bytes)
              })
              bytes. rewind()
            }
            if(! asBlockResult){
              return Some(bytes)
            } else {
              //如果 asBlockResult 为 true,调用 dataDeserialize 方法,将 bytes 数据反序列化成对象
              val values = dataDeserialize(blockId,bytes)
              if(level. deserialized){
                //在返回值之前在内存中缓存
                val putResult = memoryStore. putIterator(
                  blockId,values,level,returnValues = true,allowPersistToDisk = false)
        putResult. data match {
                  case Left(it) =>
                    return Some(newBlockResult(it,DataReadMethod. Disk,info. size))
                  case _ =>
                    throw new SparkException("Memory store did not return an iterator!")
                }
```

```
                } else {
                    return Some(newBlockResult(values,DataReadMethod.Disk,info.size))
                }
              }
            }
          }
        }
      }
      //返回 None
      None
    }
```

首先从 blockInfo 哈希表中取出 blockId 对应的 BlockInfo。如果为 Null，返回 None；如果 info 不为 null，获取 info 的 synchronize 同步块，保证只有一个线程操作该 blockInfo。为了保险，再次判断哈希表中该 BlockInfo 是否被 removeBlock 方法删除。如果该 BlockInfo 正在被其他线程写入数据，等待其他线程退出，如果等待失败，返回 None。

所有检查条件都通过了之后，根据 StorageLevel 调用对应 BlockStore 的方法，查找 blockId 对应的 Block 数据并返回。如果判断 StorageLevel 使用的是 Memory，则调用 memoryStore 的 getValues 或 getBytes 方法查询块数据并返回；如果判断 StorageLevel 使用的是 OFF_HEAP，则调用 externalBlockStore 的 getValues 或 getBytes 方法查找数据并返回；如果判断 StorageLevel 使用的是 Disk，则调用 diskStore 的 getBytes 方法查询并返回数据。如果配置了 useMemory，查询出来的数据将根据内存情况存入 MemoryStore 中，以便再次查询时直接从内存中查找，从而加快查找速度。

在本地没有查询到 blockId 对应的 Block 数据时，会调用 getRemote 方法查询远程结点上是否有该 blockId 对应的 Block 数据。接下来了解一下 getRemote 方法，源代码如下。

```
def getRemote(blockId:BlockId):Option[BlockResult] = {
    logDebug(s"Getting remote block $ blockId")
    //调用 doGetRemote 方法
    doGetRemote(blockId,asBlockResult = true).asInstanceOf[Option[BlockResult]]
}
```

getRemote 方法将通过其他结点上的 BlockManager 查询 block，将查询的结果作为 BlockResult 实例。在该方法中调用 doGetRemote 方法。源代码如下。

```
private defdoGetRemote(blockId:BlockId,asBlockResult:Boolean):Option[Any] = {
    //检查 blockId 是否为 null
    require(blockId! = null,"BlockId is null")
    //调用 BlockManagerMaster 的 getLocations 方法,获得 blockId 的位置,并使用 Random 的
  shuffle 方法,将位置打乱,这样不至于每一次都到同一个远程结点取出 blockId 对应的数据
    val locations = Random.shuffle(master.getLocations(blockId))
```

```
    //抓取失败的次数,初始为0
    var numFetchFailures = 0
    //遍历 locations
    for(loc <- locations) {
      logDebug(s"Getting remote block $blockId from $loc")
      val data = try {
        //调用 blockTransferService 的 fetchBlockSync 方法,异步抓取远程结点上的数据
        blockTransferService.fetchBlockSync(
          loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer()
      } catch {
      //异常处理
      …
      }
     //如果返回的数据不为 null
      if(data != null) {
        //如果 asBlockResult 为 true
        if(asBlockResult) {
          //返回数据,退出循环
          return Some(newBlockResult(
            //调用 dataDeserialize 方法反序列化 data
            dataDeserialize(blockId, data),
            DataReadMethod.Network,
            data.limit()))
        } else {
      //返回数据,退出循环
          return Some(data)
        }
      }
      logDebug(s"The value of block $blockId is null")
    }
    logDebug(s"Block $blockId not found")
    None
  }
```

doGetRemote 方法首先会检查 blockId 是否为空，如果为空，则会抛出以“BlockId is null”为错误信息的 IllegalArgumentExeception 异常。如果存在 blockId，调用 BlockManagerMaster 的 getLocations 方法，返回存有该 blockId 的 BlockManagerId 信息。使用 Random.shuffle 方法得到 BlockManagerId 列表，以保证请求的负载均衡。

在 for 循环中，使用 blockTransferService 的 fetchBlockSync 方法异步抓取远程 BlockManager 上对应的 blockId 数据，该方法可能会失败，失败后将打印失败信息，并转至下一次循环，直到找到数据并返回。如果没有查找到 blockId 对应的数据，则返回 None。

6.2.3 Partition 与 Block 的对应关系

至此，已了解了 Spark Storage 模块存储层的架构及代码实现。目前为止，也许读者还有一个疑问，RDD 中的 partition 是如何与 Storage 中的 blockId 对应的呢？接下来将揭开这个谜。

在 RDD 中，计算的重要方法是 iterator，该方法的源代码如下。

```
final def iterator(split:Partition,context:TaskContext):Iterator[T] = {
    //判断 StorageLeve 不为 NONE
    if(storageLevel! = StorageLevel. NONE){
      调用 cacheManager 的 getOrCompute 方法计算
      SparkEnv. get. cacheManager. getOrCompute(this,split,context,storageLevel)
    } else {
      //调用 computeOrReadCheckpoint 方法,从 checkpoint 中得到数据
      computeOrReadCheckpoint(split,context)
    }
  }
```

iterator 方法中，首先判断 StorageLevel 是否设置，如果没有设置，说明在 DiskStore、MemoryStore 和 ExternalBlockStore 中没有存储该 RDD。反之，如果设置了 StorageLevel，即 StorageLevel 不等于 StorageLevel. NONE，则说明在 BlockStore 中存有该 RDD。RDD 作为 Spark 中最重要的抽象数据集，其数据可以通过计算得到，为了避免重复计算，可以将多次使用的 RDD 缓存起来，再次使用时直接从缓存中获取，而不再重新计算，提高整体性能。cacheManager 就是一个 RDD 缓存的管理类，在其 getOrCompute 函数中，将看到 Partition 和 blockId 之间的对应关系。getOrCompute 函数的源代码如下所示。

```
def getOrCompute[T](
      rdd:RDD[T],
      partition:Partition,
      context:TaskContext,
      storageLevel:StorageLevel):Iterator[T] = {
    …
    //RDDBlockId 建立 RDD 与 block 之间的联系
    val key = RDDBlockId(rdd. id,partition. index)
    …
        }
```

代码中使用 rdd. id 和 partition. index 构建出一个 RDDBlockId 对象，在构建 RDDBlockId 对象的过程中，重写 name 方法，该方法的代码如下所示。

```
case class RDDBlockId(rddId:Int,splitIndex:Int)extends BlockId {
  override def name:String = "rdd_" + rddId + "_" + splitIndex
}
```

如上面代码所示，此时的 blockId 用传入的 rddId 和 splitIndex 拼凑而成，格式如"rdd"_ + rddId + "_" + splitIndex 格式。调用 blockManager 的 get 方法取出 blockId 对应的数据。如果 blockManager 的 get 方法返回 None，说明 BlockManager 中没有这个 block，需要通过计算获取，并将计算得到的 block 数据存入 BlockManager 中。block 的计算和存储是阻塞的，其他的线程需要等到该 block 装载结束后才能操作该 block。

RDD 中的 Transaction 操作和 Action 操作，虽然逻辑上是在 Partition 上进行的操作，但最终还是转换成了 Block，实际上对数据的存储都发生在 Block 上。

本节对 Storage 模块的通信层和存储层进行了较为全面的讲解，包括 Storage 模块通信层的通信模式，以及存储层中不同的存储实现。并且讲解了 RDD 中针对 partition 的计算最终都转化成对 Block 的计算，通过代码跟踪的方式解释了 RDD 中 Partition 和 Block 之间的关系。实际上 Partition 和 Block 是等价的，只是看待的角度不同。

在下一节中，将对比不同 Storage Level 的性能，并对如何正确选择 StorageLevel 等级进行探讨。

Section 6.3 不同 Storage Level 对比

Spark Storage 模块中设置了不同的存储级别，不同的存储级别在性能和容错性上各有优势。比如 MemoryStore 在访问速度上更快，但是由于数据都存放在 JVM 内存中，会占用大量的 JVM Storage 内存，严重情况下可能会出现 OOM。DiskStore 在访问速度上没有 MemoryStore 快，因为其需要读写磁盘，频繁的 I/O 操作会大大地降低系统性能，但是 DiskStore 能够存储的文件的大小比 MemoryStore 大得多。ExternalBlockStore 采用 JVM 外部的存储系统，ExternalBlockStore 的典型运用就是 Tachyon 内存分布式文件系统。

在本节中，将对比分析不同 StorageLevel 在性能方面的对比，以及 StorageLevel 存储级别的选择。首先看一下内存级别的存储，StorageLevel 在内存存储上提供了 4 个选择，分别是 MEMORY_ONLY、MEMORY_ONLY_2、MEMORY_ONLY_SER 和 MEMORY_ONLY_SER_2。接下来分别对这 4 种存储级别进行对比。

MEMORY_ONLY，从名称上就可以知道，只启用内存存储并且在内存中只保留一份数据。这种模式最大的缺点是数据以对象的形式存储在内存中，当数据量过大时容易造成 OOM；MEMORY_ONLY_2，这种存储级别将数据存放在内存中，并在远程结点上存储一份数据的副本，其优点是当本地结点数据被破坏时，远程结点还有一个冗余的备份，增加了抵抗风险的能力。缺点是数据仍然以对象形式存放在内存中，大量的对象容易导致 OOM，并且在远程结点冗余备份，增加了网络开销；MEMORY_ONLY_SER，这种存储模式将数据序列化之后以字节数组形式存放在内存中，大大降低了内存的消耗，但是当数据量超出 JVM 分配的内存空间时，仍然会出现 OOM；MEMORY_ONLY_SER_2，这种存储模式在序列化数据的同时，在远程结点冗余备份数据，增强了抵抗风险的能力，数据量过大也会出现 OOM，并且冗余备份还增加了网络开销。

现在的集群，虽然内存都已经很大了，但是即便再大的内存，也不能保证一定不会出现 OOM。万事都有利有弊，内存存储在速度上的确非常快，但跟磁盘比起来，其容量仍然有

限，在期盼物理内存在将来越来越大的同时，也需要采取其他的补救措施。接下来看一下磁盘存储。

StorageLevel 中以 DISK_ONLY 代表选择磁盘存储，计算的数据将通过 I/O 操作存储到磁盘上，磁盘相对于内存，其优点是有相对充分的空间对数据进行存储，当然其缺点是访问速度慢。DISK_ONLY_2 表示使用磁盘存储，并在远程结点上进行冗余备份，增加了抵抗风险的能力。纯的 MEMORY 和纯的 DISK 存储都有其优缺点，能不能取两者的长补两者的短呢？答案当然是可以，只需同时使用内存和磁盘存储即可，内存提供高速的访问速度，磁盘提供相对充分的存储空间。

在 StorageLevel 中，内存和磁盘有 4 种不同的组合方式，分别是 MEMORY_AND_DISK、MEMORY_AND_DISK_2、MEMORY_AND_DISK_SER 和 MEMORY_AND_DISK_SER。MEMORY_AND_DISK 这种存储方式表示，首先将数据放在内存，当内存不能放下该 block 数据时，将其溢出到磁盘中；MEMORY_AND_DISK_2 表示在本地和远程结点上分别保存；MEMORY_AND_DISK_SER 表示序列化存储到内存，当内存不足时，溢出到磁盘；MEMORY_AND_DISK_SER_2 在 MEMORY_AND_DISK 基础上增加了远程结点的冗余备份。

虽然磁盘和内存存储级别都可以单独提供存储服务，但其各有优缺点，磁盘和内存的配合使用克服了单独使用磁盘和内存存储的缺点。有没有第三方专门提供存储服务呢？肯定有，那就是 OFF_HEAP，OFF_HEAP 表示堆外存储，数据存储在 JVM 外边，其优点是让第三方服务专门管理数据，而 JVM 中只负责计算，将存储和计算模块分离，JVM 中少了存储，这就减少了 JVM 进行 GC 的可能，JVM GC 是相当费时的操作。Tachyon 提供堆外存储服务，Spark 中目前默认使用的堆外存储是 Tachyon。

Section 6.4 Executor 内存模型

Spark 实际上是运行于多 JVM 之上的分布式计算框架，各个 JVM 通过 RPC 框架通信。既然是 JVM，那 JVM 的堆栈空间大小就可以通过参数配置。Spark Executor 中默认 JVM 堆大小是 1 GB。在每一个 JVM 中，需要分配空间用于缓存数据，还需要分配一定空间用于计算，对于 JVM 的运行还需要预留部分空间。那么 Spark 中怎样管理 JVM 中内存分配的呢？

实际上 Spark 在 JVM 内存管理上建有一套模型。该模型图 6-4 所示：

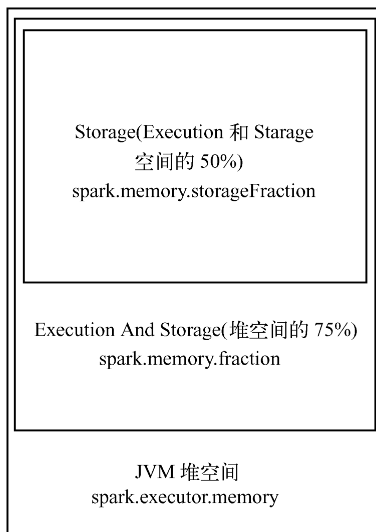


图 6-4 Executor JVM 内存模型

从图中可以看到，Spark 启动的 Executor 默认的堆大小是 1 GB，当然这个默认堆大小可以通过 spark.executor.memory 来配置。

为了安全考虑同时避免 OOM，Spark JVM 中允许使用的内存空间占 JVM 堆空间的 75%，该比例可以通过 spark. memory. fraction 来配置。例如通过 spark. executor.

memory 配置 JVM 内存大小为 1024 MB，那么安全的阀值就是 1024 * 0. 75 MB。

Spark 作为一个内存计算框架，可以在内存中缓存数据。Spark 通过 LRU 算法将数据缓存到内存中，这样 Spark JVM 中就有大量的内存被用来缓存数据。在 Spark 中用于缓存数据的内存大小是有限制的。该值默认占 Executor And Storage 堆空间的 50%，可以通过 spark. memory. storageFraction 配置该比例。按照默认值为 50%，则用于缓存数据的空间大小为 1024 * 0. 75 * 0. 5 MB。如果该值设置得过大，那么计算内存占的比例就会变少，这将可能使任务更加频繁的溢出到磁盘。

Executor 内存的大小，和性能本身当然并没有直接的关系，但是几乎所有运行时性能相关的内容都或多或少间接和内存大小相关。这个参数最终会被设置到 Executor 的 JVM 的 heap 尺寸上，对应的就是 Xmx 和 Xms 的值。

理论上 Executor 内存当然是多多益善，但是实际受机器配置，以及运行环境，资源共享，JVM GC 效率等因素的影响，还是有可能需要为它设置一个合理的大小。多大算合理，要看实际情况。

Executor 的内存基本上是 Executor 内部所有任务共享的，而每个 Executor 上可以支持的任务的数量取决于 Executor 所管理的 CPU Core 资源的多少，因此你需要了解每个任务的数据规模的大小，从而推算出每个 Executor 大致需要多少内存即可满足基本的需求。

如何知道每个任务所需内存的大小呢？很难统一的衡量，因为除了数据集本身的开销，还包括算法所需各种临时内存空间的使用，而根据具体的代码算法等不同，临时内存空间的开销也不同。但是数据集本身的大小，对最终所需内存的大小还是有一定的参考意义的。

通常来说每个分区的数据集在内存中的大小，可能是其在磁盘上源数据大小的若干倍（不考虑源数据压缩，Java 对象相对于原始数据也还要算上用于管理数据的数据结构的额外开销），需要准确的知道大小的话，可以将 RDD cache 在内存中，从 BlockManager 的 Log 输出可以看到每个 Cache 分区的大小（其实也是估算出来的，并不完全准确）反过来说，如果你的 Executor 的数量和内存大小受机器物理配置影响相对固定，那么你就需要合理规划每个分区任务的数据规模，例如采用更多的分区，用增加任务数量（进而需要更多的批次来运算所有的任务）的方式来减小每个任务所需处理的数据大小。

如前面所说 spark. executor. memory 决定了每个 Executor 可用内存的大小，而 spark. memory. storageFraction 则决定了在这部分内存中有多少可以用于 Memory Store 管理 RDD Cache 数据，剩下的内存用来保证任务运行时各种其他内存空间的需要。

spark. memory. storageFraction 默认值为 0. 5，官方文档建议这个比值不要超过 JVM Old Gen 区域的比值。这也很容易理解，因为 RDD Cache 数据通常都是长期驻留内存的，理论上也就是说最终会被转移到 Old Gen 区域（如果该 RDD 还没有被删除的话），如果这部分数据允许的尺寸太大，势必把 Old Gen 区域占满，造成频繁的 FULL GC。

如何调整这个比值，取决于你的应用对数据的使用模式和数据的规模，粗略的来说，如果频繁发生 Full GC，可以考虑降低这个比值，这样 RDD Cache 可用的内存空间减少（剩下的部分 Cache 数据就需要通过 Disk Store 写到磁盘上了），会带来一定的性能损失，但是腾出更多的内存空间用于执行任务，减少 Full GC 发生的次数，反而可能改善程序运行的整体性能。

Section
6.5 Tachyon

在前面章节中，讲解了 BlockStore 的几种不同实现，其中 ExternalBlockStore 用于将 BlockManager 中的 Block 保存到外部存储中。Spark 中默认使用的外部存储系统是 Tachyon，在 ExternalBlockStore 中，将会调用 TachyonBlockManager 中的相关方法，对 Tachyon 中的数据进行存取操作。在本节中，将介绍 Tachyon 在 Spark 中的使用及相关的 API。

6.5.1 Tachyon 简介

Tachyon（意为超光速粒子）是以内存为中心的分布式文件系统（2016 年 2 月更名为 Alluxio），拥有高性能和容错能力，能够为集群框架（如 Spark、MapReduce 等）提供可靠的内存级速度的文件共享服务。从软件栈的层次来看，Tachyon 是位于现有大数据计算框架和大数据存储系统之间的独立的一层。它利用底层文件系统作为备份，对于上层应用来说，Tachyon 就是一个分布式文件系统。

Tachyon 诞生于 UC Berkeley 的 AMPLab，其最初出现是为了解决如下问题。

- 大数据分析流水线中，数据共享通过基于磁盘文件系统（HDFS 等）性能比较缓慢。
- 大数据计算引擎的处理进程（如 Spark 的 Executor，MapReduce 的 Child JVM 等）崩溃出错后，缓存的数据也会全部丢失。
- 基于内存的系统存储数据冗余，对象太多会导致 Java GC 时间过长。

Tachyon 在大数据分析软件栈（Berkeley Data Analytics Stack）中的所属层次如图 6-5 所示。



图 6-5　Tachyon 在大数据软件栈中的地位

从图 6-5 中可见，Tachyon 是架构在最底层的分布式文件存储和上层的各种计算框架之间的一种中间件，主要职责是将那些不需要落地到磁盘里的文件落地到分布式内存文件系统中，以共享内存，从而提高效率。同时可以减少内存冗余、GC 时间等。

Tachyon 的架构是传统的 Master—Slave 架构，为了防止单点故障，可以使用 Zookeeper 做 HA。Tachyon 中使用 RamDisk（内存磁盘）技术来提高对文件的访问速度，Tachyon 中 Master 和 Worker 直接的通信协议是 Thrift。

Tachyon 技术在业界引起了广泛的关注，其开源社区也越来越活跃。在本节中，将以 Tachyon 最新版本 0.8.2 来讲解，因为在 Spark 1.6.＊中，Tachyon 采用的就是 0.8.2 版本。

6.5.2 Tachyon API 的使用

要使用 Tachyon 内存文件系统，首先要使用 TachyonFileSystemFactory 得到一个 Tachyon 的操作客户端，通过客户端进行文件的读写操作。得到 Client 客户端的代码如下。

```
TachyonFileSystem tfs = TachyonFileSystemFactory.get();
```

通过 TachyonFileSystemFactory 的 get 方法，返回一个 TachyonFileSystem 对象，通过该对象，对文件进行存储操作。接下来讲解 Tachyon 中对文件的存取操作。

1. 创建文件

通过 TachyonFileSystemFactory 对象创建文件也是十分简单，就跟磁盘文件一样。首先通过 TachyonURI 创建一个 Tachyon 文件路径，并使用文件的输出流向文件写入数据。代码如下。

```
//得到 TachyonFileSyste
TachyonFileSystem tfs = TachyonFileSystemFactory.get();
//新建 TachyonURI 路径
TachyonURI path = new TachyonURI("/myFile");
//根据 TachyonURI 创建文件并得到文件的输出流
FileOutStream out = tfs.getOutStream(path);
//写入数据
out.write("hello tachyon!");
//关闭文件系统
out.close();
```

2. 指定非默认值

在 Tachyon 所有的操作中，都有一个额外的选项可以用，它允许用户指定非默认的设置。比如可以通过这个选项设置非默认的数据块大小。代码如下。

```
//得到 TachyonFileSystem
TachyonFileSystem tfs = TachyonFileSystemFactory.get();
//创建 TachyonURI
TachyonURI path = new TachyonURI("/myFile");
//通过 OutStreamOptions 设置块大小为 128 MB
OutStreamOptions options = new OutStreamOptions.Builder(ClientContext.getConf()).setBlock-
Size(128 * Constants.MB).build();
FileOutStream out = tfs.getOutputStream(path,options);
```

3. I/O 选项

Tachyon 中使用了两种不同的存储类型：Tachyon 管理的存储和底层存储。Tachyon 管理的存储是分配给 Tachyon Workers 的内存、SSD 和 HDD，而底层存储是指被 S3、HDFS 和 Swift 等管理的存储资源。用户可以通过 TachyonStorageType 和 UnderStorageType 两个配置选项指定 Tachyon 本地存储和底层存储的合作方式。表 6-1 所示是 TachyonStorageType 和 UnderStorageType 不同的组合方式。

表 6-1　TachyonStorageType 和 UnderStorageType 不同组合方式

TachyonStorageType	UnderStorageType	组合结果	读数据	写数据
PROMOTE	NO_PERSIST	MUST_CACHE, CACHE_PROMOTE	读取数据并移到置换的最高层，并在 Tachyon Worker 中创建副本	数据被同步写入 Tachyon Worker
PROMOTE	SYNC_PERSIST	CACHE_THROUGH, CACHE_PROMOTE	读取数据并移到置换的最高层，并在 Tachyon Worker 中创建副本	数据被同步写入 Tachyon worker 和底层存储系统
STORE	NO_PERSIST	MUST_CACHE, CACHE	对于每一个完整的数据块，在 Tachyon Worker 中创建副本	数据被同步写入 Tachyon worker，不写入底层存储系统
STORE	SYNC_PERSIST	CACHE_THROUGH, CACHE	对于每一个完整的数据块，在 Tachyon Worker 中创建副本	数据被同步写入 Tachyon worker 和底层存储系统
NO_STORE	SYNC_PERSIST	THROUGH, NO_CACHE	先从 Tachon 读取，如果没有再从底层存储系统读，不创建副本	数据被同步写入底层存储系统

TachyonStorageType 和 UnderStorageType 的不同组合，将产生出不同的读、写数据策略。

4. 打开 TachyonFile

在创建好 Tachyon 文件的前提下，可用通过 open 方法打开该 Tachyon 文件。open 方法返回创建好的 TachyonFile 的引用。代码如下。

```
//得到 TachyonFileSystem 对象
TachyonFileSystem tfs = TachyonFileSystemFactory.get();
//确保/myFile 在 Tachyon 中存在
TachyonURI path = new TachyonURI("/myFile");
//打开 Tachyon 文件
TachyonFile file = tfs.open(path);
```

5. 读取数据

Open 方法打开 Tachyon 文件操作其实只返回一个 TachyonFile 的引用，要想得到文件中的内容，还需要得到该文件上的输入流，通过输入流读取 TachyonFile 文件中的内容。代码如下。

```
TachyonFileSystem tfs = TachyonFileSystemFactory.get();
TachyonURI path = new TachyonURI("/myFile");
TachyonFile file = tfs.open(path);
//打开文件,获得文件的输入流
```

```
FileInStream in = tfs. getInStream(file);
//读取数据
in. read(...);
//关闭文件,释放锁
in. close();
```

更多 API 的使用，请查看 Tachyon （Alluxio） 官方网站：http://www. alluxio. org。在了解了 Tachyon 基本的 API 使用之后，下面介绍 Tachyon 在 Spark 中的应用。

6.5.3 Tachyon 在 Spark 中的使用

在上一小节中，讲解了 BlockStore 的实现之一 ExternalBlockStore，在 ExternalBlockStore 中通过使用 ExternalBlockManager 来操作 Spark 内存中的数据，可以看到 ExternalBlockManager 是一个抽象类，目前该类的唯一实现是 TachyonBlockManager，包路径是 org. apache. spark. storage. TachyonBlockManager。在 TachyonBlockManager 中，提供了对 Tachyon 进行存取数据操作的接口。

在 TachyonBlockManager 的 init 方法中，通过传入的 blockManager 和 executorId，找到 Tachyon 的 master 地址和存放文件的根路径。并且通过 TachyonFS. get(new TachyonURI(master),new TachyonConf())的方法调用，返回一个 TachyonFs 文件系统句柄，通过该文件系统句柄就可以方便地存取文件了。值得注意的是，在 Tachyon 0. 8. 2 中 TachyonFs 类已经过时，要得到 Tachyon 文件系统的引用，需要使用 TachyonFileSystem. get 方法。

下面分别查看 TachyonBlockManager 中的源代码，了解 Spark 中是如何使用 Tachyon 进行数据读写的。

1. TachyonBlockManager 向 Tachyon 中存放数据

Spark 中的数据要存入 Tachyon，需要通过 TachyonBlockManager 提供的方法，这里以 putBytes 方法为例，讲解 Spark 中是如何使用 Tachyon 存放数据的。putBytes 方法存放 BlockId 对应的字节数组中的数据，putBytes 方法的源代码如下。

```
override def putBytes(blockId:BlockId,bytes:ByteBuffer):Unit = {
    //调用 getFile 方法,以 blockId 中的 name 属性作为哈希函数的参数,映射得到 Tachyon 对应的 TachyonFile
val file = getFile(blockId)
    //打开文件上的输入流
    val os = file. getOutStream(WriteType. TRY_CACHE)
    try {
      //向输入流中写入字节数组数据
      os. write(bytes. array())
    } catch {
       //捕获异常
       case NonFatal(e) =>
          logWarning(s"Failed to put bytes of block $ blockId into Tachyon",e)
```

```
            //关闭输入流
              os. cancel( )
        } finally {
            //关闭输入流
            os. close( )
        }
    }
```



putBytes 方法接受两个参数，分别是 BlockId 对象和 ByteBuffer 对象，分别代表 RDD 产生的 blockId 和 RDD 中的数据。通过 getFile（blockId）方法，取出 blockId 哈希映射对应的 TachyonFile，首先得到文件上的输出流，再将 byteBuffer 数据写入 TachyonFile 文件中。也许读者会感到奇怪，这里并没有对 Tachyon 的操作。在 putBytes 方法中没有看到任何 Tachyon 相关 API 的调用，答案在调用的 getFile 方法中。getFile 的源代码如下。

```
//getFile 方法,传入 BlockId 对象
def getFile( blockId:BlockId) :TachyonFile = getFile( blockId. name)
//getFile 方法,传入 BlockId 对象中的 name 属性
def getFile( filename:String) :TachyonFile = {
    //计算 filename 的非负哈希值
    val hash = Utils. nonNegativeHash( filename)
    //哈希值与 tachyonDirs 保存目录个数的余数作为父目录的 id
    val dirId = hash % tachyonDirs. length
    //哈希值与 tachyonDirs 长度的商再求余,得到子目录的 id
      val subDirId = ( hash / tachyonDirs. length) % subDirsPerTachyonDir
      //从二维数组中取出 dirId 和 subDirId 对应的目录
      var subDir = subDirs( dirId) ( subDirId)
      //如果目录为空
      if( subDir = = null) {
        //使用 subDirs( dirId) 上的 synchronized 同步块,防止多线程同时操作
        subDir = subDirs( dirId). synchronized {
          //得到 old 值
          val old = subDirs( dirId) ( subDirId)
          //若 old 值不为空,说明 subDirs( dirId) ( subDirId) 对应的目录不为空,直接返回 old
          if( old! = null) {
            old
          } else {
            //若 old 为空,说明目录还没有创建,构建对应目录
             val path = newTachyonURI( s" $ { tachyonDirs( dirId) }/ $ { "%02x". format( sub-
DirId) } " )
            //使用 TachyonFS 的 mkdir 方法创建 path 目录
client. mkdir( path)
            //使用 TachyonFS 的 getFile 方法,得到 path 目录对应的 TachonFile 对象
            val newDir = client. getFile( path)
```




```
                //将新目录 newDir 赋值给 subDirs( ) ( dirId) ( subDirId)
                subDirs( dirId) ( subDirId) = newDir
                //返回 newDir
                newDir
            }
        }
    }
    //以 subDir 和 fileName 构建出 file 的全路径,创建出 TachyonURI 对象
    val filePath = new TachyonURI( s" $ subDir/ $ filename" )
    //使用 TachyonFs 的 exist 方法检测 filePath 是否存在
    if( ! client. exist( filePath) ) {
      //若 filePath 不存在,则使用 TachyonFs 的 createFile 方法创建 filePath 对应的 TachyonFile
      client. createFile( filePath)
    }
    //使用 TachyonFs 的 getFile 方法,返回 TachyonURI 对应的 TachyonFile 对象
    val file = client. getFile( filePath)
    file
}
```

可以看到 getFile 方法返回 TachyonFile 对象。代码中首先通过 fileName 经哈希函数求出 fileName 的非负哈希值，映射找出对应的目录及子目录。使用 TachyonFs 查看对应的目录及子目录是否存在，如果不存在子目录，则调用 TachyonFs 的 mkdir 方法创建子目录。得到文件的完整路径后，先通过 client. exist 方法判断 filename 对应的文件是否存在，如果不存在，则调用 createFile 方法创建文件，否则直接通过 getFile 方法返回该 TachyonFile。从这个方法中，清楚地看到了在 Spark 中是如何使用 Tachyon API 来保存数据到分布式内存文件系统中的，相信这对于想单独使用 Tachyon 来作为共享内存系统的读者来说，有非常大的借鉴意义。

在使用 Tachyon 时，其实可以借鉴 Spark 中 Tachyon 的使用方式来构建自己的内存数据共享应用。介绍完 Spark 中数据保存 Tachyon 的方法后，接下来看一下 Spark 中是如何从 Tachyon 中取出数据的。

2. TachyonBlockManager 从 Tachyon 中取出数据

Spark 中取出 Tachyon 中的数据也很简单，这里以 getBytes 为例，看一看在 Spark 中是如何从 Tachyon 中读取数据的。getBytes 方法有一个参数，这个参数为 RDD 产生的 BlockId 对象，从 BlockId 对象中取出 name 属性，该属性在 getFile 方法中，经由哈希函数映射取出对应的 TachyonFile 文件。getBytes 方法的代码如下。

```
override def getBytes( blockId: BlockId) : Option[ ByteBuffer] = {
    //getFile 方法得到 Tachyon 中 blockId 对应的 TachyonFile 对象
    val file = getFile( blockId)
    //如果 TachyonFile 为 null 或者 file 的 hosts 大小为 0,直接返回 None
    if( file == null || file. getLocationHosts. size == 0) {
      return None
```

```
    }
    //得到 TachyonFile 上的输入流
    val is = file.getInStream(ReadType.CACHE)
     try {
        //文件的长度
        val size = file.length
        //构建 size 大小的字节数组
        val bs = new Array[Byte](size.asInstanceOf[Int])
        //使用 ByteStreams 的 readFully 方法,从输入流将字节信息读入 bs 字节数组中
        ByteStreams.readFully(is,bs)
        //将字节数组包装成 ByteBuffer 对象
        Some(ByteBuffer.wrap(bs))
     } catch {
        //捕获异常,打印错误,返回 None
        case NonFatal(e) =>
          logWarning(s"Failed to get bytes of block $blockId from Tachyon",e)
          None
     } finally {
        //关闭输入流
        is.close()
     }
  }
```

getBytes 方法接受一个参数 BlockId，和 putBytes 一样，需要调用 getFile（blockId）方法，返回 blockId 经哈希映射所对应的 TachyonFile。首先判断 getFile 方法的返回值，如果该返回值为空或 file.getLocationHosts 的大小为 0，直接返回 None；否则得到该文件的输入流，并通过输入流读取数据到数组中，返回读取到的数据，最后关闭输出流。

Tachyon 作为一个分布式内存文件系统，向外提供了方便的 API，通过这些 API 可以很方便地整合 Tachyon 到其他系统中，有关 Tachyon 的更多信息可以到 Tachyon（Alluxio）的官网查看，网址为：http://www.alluxio.org。

Section 6.6 小结

本章主要围绕 Storage 模块中的通信层和存储层进行讲解，在 6.1 节讲解了 Storage 的定义、原理及内部使用的设计模式；6.2 节是本章最重要的两个小节，在 6.2.1 讲解了 Storage 模块的通信层，6.2.2 讲解了 Storage 模块的存储层。

Spark 预先定义了 StorageLevel，在使用的时候可以直接通过 StorageLevel 选择数据存储的级别。当然不同的存储级别对性能及硬件设施都有影响，因此在 6.3 小节详细对比了 StorageLevel 的不同等级，在 6.4 小节论述了 StorageLevel 对性能的影响，在 6.5 小节简单介绍了目前 Spark 中存储级别为 OFF_HEAP 的唯一实现——Tachyon，简单介绍了其 API 的使用，并通过源代码介绍 Tachyon 在 Spark 中的使用。

第 7 章　Shuffle 机制

在大数据计算框架中，Shuffle 阶段（Phase）的设计优劣是决定性能好坏的关键因素之一。为了深入理解 Shuffle 阶段的各个细节，并进一步在理解的基础上优化代码，减少不必要的 Shuffle 开销，本章将深入 Spark Shuffle 阶段的源代码实现，详细解析 SparkShuffle 阶段的实现细节，主要内容包括 Shuffle 框架的演进史、Shuffle 框架的可插拔设计、Shuffle 阶段的数据读写设计，以及当前 Spark 中已经支持的 Shuffle 阶段的 3 种设计与实现。

Section 7.1 Shuffle 概述

Shuffle 也称为“洗牌”，从字面含义上理解，就是对数据进行重组。在 Master 调度机制的实现源代码中，也可以看到 Shuffle 在数据重组上的一个应用场景，代码如下。

```
1.  // Drivers 在 Executors 的分配上具有优先权
2.
3.  //对当前可分配的 Worker 进行洗牌,可以帮助 Drivers 的均衡部署
4.  val shuffledWorkers = Random.shuffle(workers)      // Randomization helps balance drivers
5.  for(worker <- shuffledWorkers if worker.state == WorkerState.ALIVE) {
```

其中第 4 行代码就是使用了随机洗牌的方式对数据进行重组，重新整理数据的顺序来实现调度均衡的需求。此处的数据重组仅仅是顺序上的，而对应在 MR 框架中，本质上是将不同的数据重新组织到拆分的小数据块中（所以在某些场景下（协同分区），如果可以避免数据重组，或避免在迭代中多次不必要的数据重组，在构建 RDD 时，可以避免使用宽依赖来进行优化），只是重组数据的过程往往伴随着磁盘 I/O 与网络 I/O 等开销。

对应在 Hadoop MapReduce 框架或 Spark 框架中，Shuffle 阶段本质上也是对数据进行重组，只是由于分布式计算的特性和要求，在实现的细节上会更加复杂。

在 MapReduce 框架中，Shuffle 阶段是连接 Map 和 Reduce 之间的桥梁，Map 阶段通过 Shuffle 过程将数据输出到 Reduce 阶段中。由于 Shuffle 涉及磁盘的读写和网络 I/O，因此 Shuffle 性能的高低直接影响整个程序的性能。Spark 本质上也是一种 MapReduce 框架，因此也会有自己的 Shuffle 过程实现。

在学习 Shuffle 的过程中，通常都会引用 Hadoop MapReduce 框架中的 Shuffle 过程作为入门或比较，同时也会引用 Hadoop MapReduce 框架的 Shuffle 过程中常用的术语，下面参考网络上描述该过程经典的框架图，如图 7-1 所示。

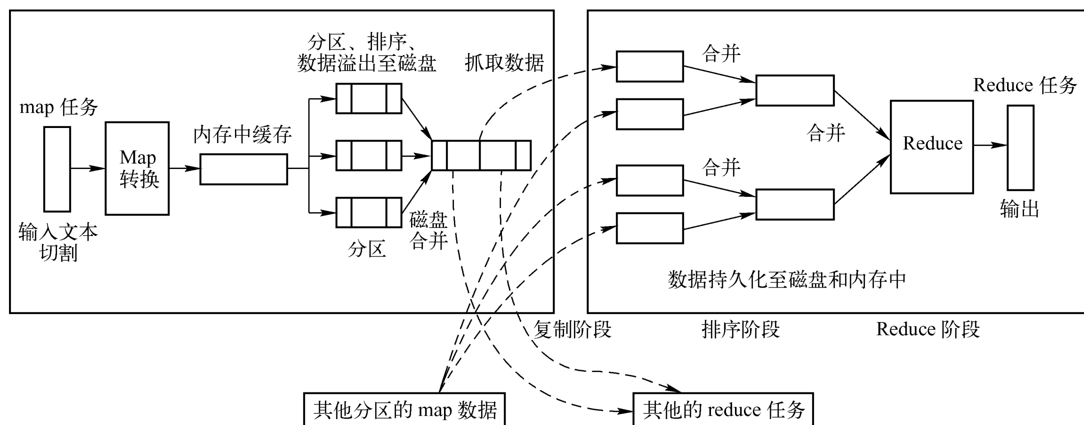


图 7-1　Hadoop MapReduce 框架中的 Shuffle 框架

其中，Shuffle 是 MapReduce 框架中的一个特定阶段，介于 Map 阶段和 Reduce 阶段之间。Map 阶段负责准备数据，Reduce 阶段则读取 Map 阶段所准备的数据，然后进一步对数据进行处理。即 Map 阶段实现 Shuffle 过程中的数据持久化（即数据写入），而 Reduce 阶段实现 Shuffle 过程中的数据读取。

在图 7-1 中，Mapper 端与 Reduce 端之间的数据交互通常都伴随着一定的网络 I/O，因此对应数据的序列化与压缩等技术也是 Shuffle 中必不可少的一部分。可以根据特定场景选取合适的序列化方式与压缩算法进行调优，在此仅给出相关的配置属性的简单描述，具体如表 7-1 所示。

表 7-1　Shuffle 序列化方式与压缩算法的配置属性

配置属性	默认值	描述
spark. serializer	org. apache. spark. serializer. JavaSerializer（当使用 Spark SQL Thrift Server 时，默认为 org. apache. spark. serializer. KryoSerializer）	序列化器，当需要在网络中传输或以序列化方式缓存时用于序列化对象所需的类
spark. shuffle. compress	true	是否对 Map 端输出文件进行压缩。为了减少网络 I/O 等，通常会使用压缩。对应的压缩算法由 spark. io. compression. codec 指定
spark. shuffle. spill. compress	true	在数据 Spill 过程中是否进行压缩的控制。对应的压缩算法由 spark. io. compression. codec 指定
spark. io. compression. codec	snappy	该 codec 用于压缩内部数据，如 RDD 分区数据，广播变量的数据及 Shuffle 的输出数据。默认情况下，Spark 提供 3 种 codecs：lz4、lzf 和 snappy。指定时也可以指定完整类名。如：org. apache. spark. io. LZ4CompressionCodec，org. apache. spark. io. LZFCompressionCodec，和 org. apache. spark. io. SnappyCompressionCodec

可以基于图 7-1 所示的框架，抽象地理解 Shuffle 阶段的实现过程，但在具体的实现上，不同的 Shuffle 设计会有不同的实现细节。

Section

7.2 Shuffle 的框架

7.2.1 Shuffle 的框架演进

Spark 的 Shuffle 框架演进历史可以从框架本身的演进和 Shuffle 具体实现机制的演进两部分进行解析。

框架本身的演进可以从面向接口编程的原则出发，结合 Build 设计模式进行理解。整个 Spark 的 Shuffle 框架是从 Spark 1.1 版本开始，提供了便于测试和扩展的可插拔式框架。

而对应在 Shuffle 的具体实现机制的演进部分，可以跟踪 Shuffle 实现细节在各个版本中的变更。具体体现在 Shuffle 数据的写入或读取，以及读写相关的数据块解析方式。下面简单描述一下整个演进过程。

在 Spark 1.1 之前，Spark 中只实现了一种 Shuffle 方式，即基于 Hash 的 Shuffle。在基于 Hash 的 Shuffle 的实现方式中，每个 Mapper 阶段的 Task 都会为每个 Reduce 阶段的 Task 生成一个文件，通常会产生大量的文件（即对应为 M×R 个中间文件，其中，M 表示 Mapper 阶段的 Task 个数，R 表示 Reduce 阶段的 Task 个数），伴随着大量的随机磁盘 I/O 操作与大量的内存开销。

为了缓解上述问题，在 Spark 0.8.1 版本中为基于 Hash 的 Shuffle 的实现引入了 Shuffle Consolidate 机制（即文件合并机制），即将 Mapper 端生成的中间文件进行合并的处理机制。通过设置配置属性“spark.shuffle.consolidateFiles”为 true，来减少中间生成的文件数量。通过文件合并，可以将中间文件的生成方式修改为每个执行单位（类似于 Hadoop 的 Slot），为每个 Reduce 阶段的 Task 生成一个文件。其中，执行单位对应为：每个 Mapper 阶段的 Core 数/每个 Task 分配的 Core 数（默认为 1）。最终可以将文件个数从 M×R 修改为 E×C/T×R，其中，E 表示 Executor 个数，C 表示可用 Core 个数，T 表示 Task 所分配的 Core 个数。

基于 Hash 的 Shuffle 的实现方式中，生成的中间结果文件的个数都会依赖于 Reduce 阶段的 Task 个数，即 Reduce 端的并行度，因此文件数仍然不可控，无法真正解决问题。因此，为了更好地解决问题，在 Spark 1.1 版本引入了基于 Sort 的 Shuffle 实现方式，并且在 Spark 1.2 版本之后，默认的实现方式也从基于 Hash 的 Shuffle 修改为基于 Sort 的 Shuffle 实现方式，即使用的 ShuffleManager 从默认的 hash 修改为 sort。首先，每个 Mapper 阶段的 Task 不会为每个 Reduce 阶段的 Task 生成一个单独的文件；而是全部写到一个数据（Data）文件中，同时生成一个索引（Index）文件，Reduce 阶段的各个 Task 可以通过该索引文件获取相关的数据。避免产生大量文件的直接收益就是降低随机磁盘 I/O 与内存的开销。最终生成的文件个数减少到 2 MB，表示每个 Mapper 阶段的 Task 分别生成两个文件，分别为数据文件与索引文件。

随着 Tungsten 计划的引入与优化，从 Spark 1.4 版本开始（Tungsten 计划目前在 Spark 1.5 与 Spark 1.6 两个版本中分别实现了第一与第二两个阶段），在 Shuffle 过程中也引入了基于 Tungsten – Sort 的 Shuffle 实现方式，通过 Tungsten 项目所做的优化，可以极大地提高 Spark 在数据处理上的性能。

为了更合理、更高效地使用内存，在 Spark 的 Shuffle 实现方式演进过程中，引进了外部排序等处理机制（针对基于 Sort 的 Shuffle 机制。基于 Hash 的 Shuffle 机制从最原始的全部放入内存改为记录级写入）。同时，为了保存 Shuffle 结果，提高性能，以及支持资源动态分配等特性，也引进了外部 Shuffle 服务等机制。

7.2.2 Shuffle 的框架内核

Shuffle 框架的设计可以从两方面去理解：一方面为了 Shuffle 模块更加内聚，并与其他模块解耦；另一方面为了更方便替换、测试和扩展 Shuffle 的不同实现方式。从 Spark 1.1 版本开始，引进了可插拔式的 Shuffle 框架（通过将 Shuffle 相关的实现封装到一个统一的对外接口，提供一种具体实现可插拔的框架）。Spark 框架中，通过 ShuffleManager 来管理各种不同实现机制的 Shuffle 过程，由 ShuffleManager 统一构建、管理具体实现子类来实现 Shuffle 框架的可插拔的 Shuffle 机制。

在详细描述 Shuffle 框架实现细节之前，先给出可插拔式 Shuffle 的整体架构的类图，如图 7-2 所示。

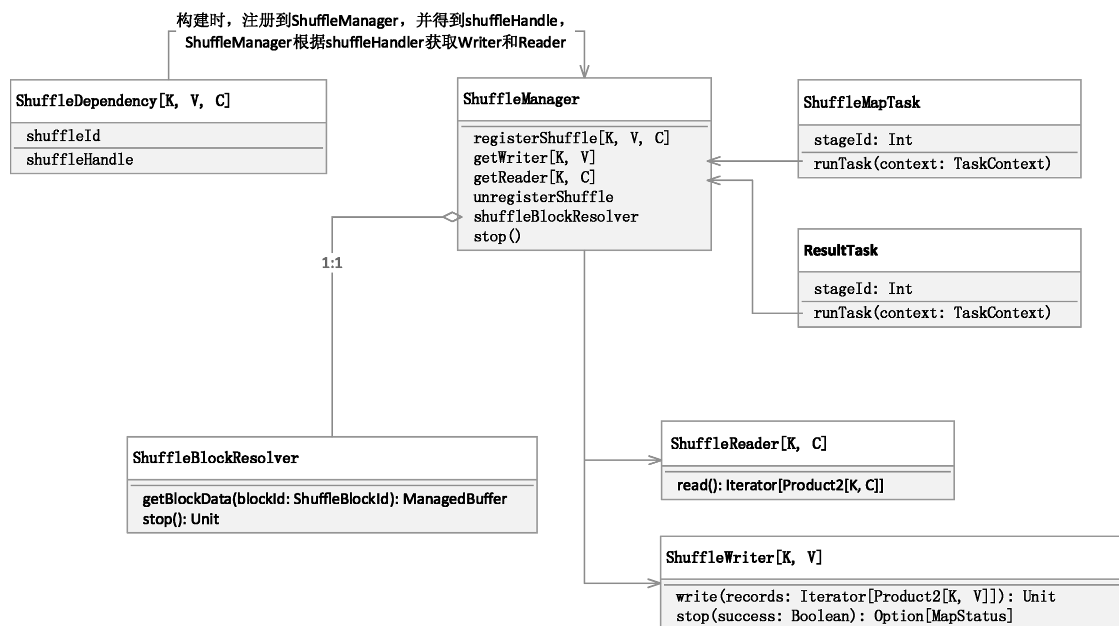


图 7-2 可插拔式 Shuffle 的整体架构的类图

在 DAG 的调度过程中，Stage 阶段的划分是根据是否有 Shuffle 过程，也就是当存在 ShuffleDependency 的宽依赖时，需要进行 Shuffle，这时会将作业（Job）划分成多个 Stage。相应的，在源代码实现中，通过在划分 Stage 的关键点——构建 ShuffleDependency 时——进行 Shuffle 注册，获取后续数据读写所需的 ShuffleHandle。

Stage 阶段划分的详细过程可以参考 DAG 调度章节的内容，最终每个作业（Job）提交后都会对应生成一个 ResultStage 与若干个 ShuffleMapStage，其中 ResultStage 表示生成作业的最终结果所在的 Stage。ResultStage 与 ShuffleMapStage 中的 Task 分别对应了 ResultTask 与

ShuffleMapTask。一个作业，除了最终的 ResultStage 外，其他若干 ShuffleMapStage 中的各个 ShuffleMapTask 都需要将最终的数据根据相应的分区器（Partitioner）对数据进行分组（即将数据重组到新的各个分区中），然后持久化分组后的数据。相应的，每个 RDD 本身记录了它的数据来源，在计算（compute）时会读取所需的数据，对于带有宽依赖的 RDD，读取时会获取在 ShuffleMapTask 中持久化的数据。

从图 7-2 中可以看到，外部宽依赖相关的 RDD 与 ShuffleManager 之间的注册交互，通过该注册，每个 RDD 自带的宽依赖（ShuffleDependency）内部会维护 Shuffle 的唯一标识信息 ShuffleId，以及与 Shuffle 过程具体读写相关的句柄 ShuffleHandle，后续在 ShuffleMapTask 中启动任务（Task）时，可以通过该句柄获取相关的 Shuffle 写入器实例，实现具体的数据磁盘写操作。

而在宽依赖（ShuffleDependency）的 RDD 中，执行 compute 时会去读取上一 Stage 为其输出的 Shuffle 数据，此时同样会通过该句柄获取相关的 Shuffle 读取器实例，实现具体数据的读取操作。需要注意的是，当前 Shuffle 的读写过程中，与 BlockManager 的交互，是通过 MapOutputTracker 来跟踪 Shuffle 过程中各个任务的输出数据的。在任务完成等场景中，会将对应的 MapStatus 信息注册到 MapOutputTracker 中，而在 compute 时的数据读取过程中，也会通过该跟踪器来获取上一 Stage 的输出数据在 BlockManager 中的位置，然后通过 getReader 得到的数据读取器，从这些位置中读取数据。

目前对 Shuffle 的输出进行跟踪的 MapOutputTracker 并没有和 Shuffle 数据读写类一样，也封装到 Shuffle 的框架中。如果从代码聚合与解耦等角度出发，也可以将 MapOutputTracker 合并到整个 Shuffle 框架中，然后在 Shuffle 写入器输出数据之后立即进行注册，在数据读取器读取数据之前获取位置等（但对应的 DAG 等调度部分也需要进行修改）。

ShuffleManager 封装了各种 Shuffle 机制的具体实现细节，其包含的接口与属性如下所示。

1）registerShuffle：每个 RDD 在构建它的父依赖（这里特指 ShuffleDependency）时，都会先注册到 ShuffleManager，获取 ShuffleHandler，用于后续数据块的读写等。

2）getWriter：可以通过 ShuffleHandler 获取数据块写入器，写数据时通过 Shuffle 的块解析器 shuffleBlockResolver 获取写入位置（通常将写入位置抽象为 Bucket，位置的选择则由洗牌的规则即 Shuffle 的分区器决定），然后将数据写入到相应位置（理论上，该位置可以位于任何能存储数据的地方，包括磁盘、内存或其他存储框架等，目前在可插拔框架的几种实现中，Spark 与 Hadoop 一样都采用了磁盘的方式进行存储，主要目的是为了节约内存，同时提高容错性）。

3）getReader：可以通过 ShuffleHandler 获取数据块读取器，然后通过 Shuffle 的块解析器 shuffleBlockResolver 来获取指定数据块。

4）unregisterShuffle：与注册相对应，用于删除元数据等后续清理操作。

5）shuffleBlockResolver：Shuffle 的块解析器，通过该解析器，为数据块的读写提供支撑层，便于抽象具体的实现细节。

7.2.3 Shuffle 框架的源代码解析

用户可以通过自定义 ShuffleManager 接口，并通过指定的配置属性进行设置，也可以通

过该配置属性指定 Spark 已经支持的 ShuffleManager 具体实现子类。

在 SparkEnv 源代码中可以看到设置的配置属性，以及当前在 Spark 的 ShuffleManager 可插拔框架中已经提供的 ShuffleManager 具体实现，代码如下。

```
//用户可以通过短格式的命名来指定所使用的 ShuffleManager
  // Let the user specify short names for shuffle managers

//下面是 3 种已经支持的 ShuffleManager,包括 hash、sort 及 tungsten - sort
val shortShuffleMgrNames = Map(
      "hash" -> "org. apache. spark. shuffle. hash. HashShuffleManager",
      "sort" -> "org. apache. spark. shuffle. sort. SortShuffleManager",
      "tungsten - sort" -> "org. apache. spark. shuffle. sort. SortShuffleManager")

//指定 ShuffleManager 的配置属性:"spark. shuffle. manager"
   //默认情况下使用 sort,即 SortShuffleManager 的实现
   val shuffleMgrName = conf. get("spark. shuffle. manager","sort")
      val shuffleMgrClass = shortShuffleMgrNames. getOrElse(shuffleMgrName. toLowerCase, shuffleMgrName)
     val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
```

从上面的代码中可以看出，ShuffleManager 是 Spark Shuffle 系统提供的一个可插拔式接口，可以通过"spark. shuffle. manager" 配置属性来设置自定义的 ShuffleManager。

在 Driver 和每个 Executor 的 SparkEnv 实例化过程中，都会创建一个 ShuffleManager，用于管理块数据，提供集群块数据的读写，包括数据的本地读写和读取远程结点的块数据。

Shuffle 系统的框架可以以 ShuffleManager 作为入口进行解析。在 ShuffleManager 中指定了整个 Shuffle 框架所使用的各个组件，包括如何注册到 ShuffleManager 以获取一个用于数据读写的处理句柄 ShuffleHandle，通过 ShuffleHandle 获取特定的数据读写接口：ShuffleWriter 与 ShuffleReader，以及如何去获取块数据信息的解析接口 ShuffleBlockResolver。下面通过源代码分别对这几个比较重要的组件进行解析。

1. ShuffleManager 源代码解析

由于 ShuffleManager 是 Spark Shuffle 系统提供的一个可插拔式接口，已经提供的具体实现子类或自定义具体实现子类时，都需要重写 ShuffleManager 类的抽象接口，下面首先分析 ShuffleManager 的源代码，代码如下。

```
package org. apache. spark. shuffle

import org. apache. spark. {TaskContext,ShuffleDependency}

/**
  * Shuffle 系统的可插拔接口。在 Driver 和每个 Executor 的 SparkEnv 实例中创建
 * Pluggable interface for shuffle systems.  A ShuffleManager is created in SparkEnv on the
```




```
  * driver and on each executor, based on the spark.shuffle.manager setting. The driver
  * registers shuffles with it, and executors (or tasks running locally in the driver) can ask to
  * read and write data.
  * NOTE: this will be instantiated by SparkEnv so its constructor can take a SparkConf and
  * boolean isDriver as parameters.
  */
private[spark] trait ShuffleManager {
  /**
 * 在 Driver 端向 ShuffleManager 注册一个 Shuffle，获取一个 Handle，
 * 在具体 Task 中会通过该 Handle 来读写数据

   */
 def registerShuffle[K, V, C](
      shuffleId: Int,
      numMaps: Int,
      dependency: ShuffleDependency[K, V, C]): ShuffleHandle

/** 获取对应给定的分区所使用的 ShuffleWriter，该方法在 Executor 上执行
  * 各个 Map 任务时调用
 * Get a writer for a given partition. Called on executors by map tasks. */
  def getWriter[K, V](handle: ShuffleHandle, mapId: Int, context: TaskContext): ShuffleWriter[K, V]

  /**
    * 获取在 Reduce 阶段读取分区的 ShuffleReader，对应读取的分区由
    * [startPartition to endPartition - 1]区间指定。该方法在 Executor 上执行
    * 各个 Reduce 任务时调用
    * Get a reader for a range of reduce partitions (startPartition to endPartition - 1, inclusive).
    * Called on executors by reduce tasks.
    */
def getReader[K, C](
     handle: ShuffleHandle,
     startPartition: Int,
     endPartition: Int,
     context: TaskContext): ShuffleReader[K, C]

/**
      * 该接口和 registerShuffle 分别负责元数据的取消注册与注册
 * 调用 unregisterShuffle 接口时，会移除 ShuffleManager 中对应的元数据信息
      * Remove a shuffle's metadata from the ShuffleManager.
      * @return true if the metadata removed successfully, otherwise false.
      */
```



```
  def unregisterShuffle(shuffleId:Int):Boolean

  /**
    * 返回一个可以基于块坐标来获取 Shuffle 块数据的 ShuffleBlockResolver

    */
  def shuffleBlockResolver:ShuffleBlockResolver

  /** 终止 ShuffleManager
* Shut down this ShuffleManager.  */
  def stop():Unit
}
```

2. ShuffleHandle 源代码解析

ShuffleHandle 比较简单，用于记录 Task 与 Shuffle 相关的一些元数据，同时也可以作为不同具体 Shuffle 实现机制的一种标志信息，控制不同具体实现子类的选择等。如下所示。

```
abstract class ShuffleHandle(val shuffleId:Int)extends Serializable {}
```

3. ShuffleWriter 源代码解析

继承 ShuffleWriter 的每个具体子类会实现 write 接口，给出任务在输出时的记录具体写的方法。如下列代码所示。

```
/**
  * Obtained inside a map task to write out records to the shuffle system.
  */
private[spark] abstract class ShuffleWriter[K,V] {
  /** Write a sequence of records to this task's output  */
  @throws[IOException]
  def write(records:Iterator[Product2[K,V]]):Unit

  /** Close this writer,passing along whether the map completed  */
  def stop(success:Boolean):Option[MapStatus]
}
```

4. ShuffleReader 源代码解析

继承 ShuffleReader 的每个具体子类会实现 read 接口，计算时负责从上一阶段 Stage 的输出数据中读取记录。如下列代码所示。

```
private[spark] traitShuffleReader[K,C] {
  /** Read the combined key-values for this reduce task  */
  def read():Iterator[Product2[K,C]]

  /**
    * Close this reader.
```

```
     * TODO:Add this back when we make theShuffleReader a developer API that others can implement
     * (at which point this will likely be necessary).
     */
  // def stop():Unit
}
```

5. ShuffleBlockResolver 源代码解析

ShuffleBlockResolver 的源代码如下所示。

```
/**
 * 该特质的具体实现子类知道如何通过一个逻辑 Shuffle 块标识信息来获取一个
 * 块数据。具体实现可以使用文件或文件段来封装 Shuffle 的数据。这是获取 Shuffle 块
  * 数据时所使用的抽象接口,在 BlockStore 中使用




  */
trait ShuffleBlockResolver {
  typeShuffleId = Int

  /**
    * 获取指定块的数据。如果指定块的数据无法获取,则抛出异常


    */
def getBlockData(blockId:ShuffleBlockId):ManagedBuffer

  def stop():Unit
}
```

继承 ShuffleBlockResolver 的每个具体子类会实现 getBlockData 接口，给出具体的获取块数据的方法。

目前在 ShuffleBlockResolver 的各个具体子类中，除了给出获取数据的接口之外，通常会提供如何解析块数据信息的接口，即提供了写数据块时的物理块与逻辑块之间映射关系的解析方法。

7.2.4 Shuffle 的注册

在图 7-2 中可以看到，当构建一个宽依赖（ShuffleDependency）的 RDD 时，该 RDD 需要向 ShuffleManager 注册。之所以在构建宽依赖时注册，其原因在于 DAG 调度器中的 Stage

是根据宽依赖进行划分的，而对应的宽依赖类目前仅有 ShuffleDependency。

对应图 7-2 中与注册相关的具体代码如下。

```
class ShuffleDependency[K:ClassTag,V:ClassTag,C:ClassTag](
    @transient private val _rdd:RDD[_ <: Product2[K,V]],
    val partitioner:Partitioner,
    val serializer:Option[Serializer] = None,
    val keyOrdering:Option[Ordering[K]] = None,
    val aggregator:Option[Aggregator[K,V,C]] = None,
    val mapSideCombine:Boolean = false)
  extends Dependency[Product2[K,V]] {

//唯一标识信息,可以看到,是通过 rdd 的上下文去获取的,
  //因此针对特定的 rdd,每个 shuffleId 值都是唯一的
val shuffleId:Int = _rdd.context.newShuffleId()

  //获取 ShuffleHandle 实例,后续获取 Shuffle 写入器与读取器时需要
 val shuffleHandle:ShuffleHandle = _rdd.context.env.shuffleManager.registerShuffle(
    shuffleId,_rdd.partitions.size,this)

  // Shuffle 数据清理器的设置,可以扩展到当使用外部 Shuffle 服务时,数据如何清理等
  _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

在代码中可以看到，唯一标识信息 shuffleId 是通过 RDD 的上下文来获取的；另外 ShuffleManager 也是在当前 SparkEnv 的实例中获取，然后注册到 SparkEnv 中的 ShuffleManager 实例。

7.2.5 Shuffle 读写数据的源代码解析

1. Shuffle 写数据的源代码解析

从 Spark Shuffle 的整体框架中可以看到，在 ShuffleManager 中提供了 Shuffle 相关数据块的写入与读取，即对应的接口 getWriter 与 getReader。

在解析 Shuffle 框架数据读取过程中，可以构建一个具有 ShuffleDependency 的 RDD，查看在执行过程中，Shuffle 框架中的数据读写接口 getWriter 与 getReader 如何使用，通过这种具体案例的方式来加深对源代码的理解。

Spark 中 Shuffle 具体的执行机制可以参考本书的其他章节，在此仅分析与 Shuffle 直接相关的内容。通过 DAG 调度机制的解析可以知道，Spark 中的一个作业可以根据宽依赖切分 Stage，而在 Stage 中，相应的 Task 也包含两种，即 ResultTask 与 ShuffleMapTask。其中，一个 ShuffleMapTask 会基于 ShuffleDependency 中指定的分区器将一个 RDD 的元素拆分到多个 bucket 中，此时通过 ShuffleManager 的 getWriter 接口来获取数据与 bucket 的映射关系。而

ResultTask 对应的是一个将输出返回给应用程序 Driver 端的 Task，在该 Task 执行过程中，最终都会调用 RDD 的 compute 对内部数据进行计算，而在带有 ShuffleDependency 的 RDD 中，在 compute 计算时，会通过 ShuffleManager 的 getReader 接口获取上一个 Stage 的 Shuffle 输出结果来作为本次 Task 的输入数据。

首先查看 ShuffleMapTask 中的数据写流程，具体代码如下。

```
override def runTask(context:TaskContext):MapStatus = {
   …
//首先从 SparkEnv 获取 ShuffleManager
//然后从 ShuffleDependency 中获取注册到 ShuffleManager 时所得到的 shuffleHandle
//根据 shuffleHandle 和当前 Task 对应的分区 ID,获取 ShuffleWriter
//最后根据获取的 ShuffleWriter,调用其 write 接口,写入当前分区的数据
var writer:ShuffleWriter[Any,Any] = null
    try {
      val manager = SparkEnv.get.shuffleManager
      writer = manager.getWriter[Any,Any](dep.shuffleHandle,partitionId,context)
      writer.write(rdd.iterator(partition,context).asInstanceOf[Iterator[_ <:Product2[Any,
      Any]]])
      writer.stop(success = true).get
    } catch {
      …
    }
  }
```

2. Shuffle 读数据的源代码解析

对应的数据读取器，从 RDD 的 5 个抽象接口可知，RDD 的数据流最终会经过算子操作，即 RDD 中的 compute 方法。下面以包含宽依赖的 RDD、CoGroupedRDD 为例，查看如何获取 Shuffle 的数据。具体代码如下。

```
//对指定分区进行计算的抽象接口,以下为 CoGroupedRDD 具体子类中该方法的实现
override def compute(s:Partition,context:TaskContext):Iterator[(K,Array[Iterable[_]])] = {
    val split = s.asInstanceOf[CoGroupPartition]
    val numRdds = dependencies.length

    // A list of(rdd iterator,dependency number)pairs
    val rddIterators = new ArrayBuffer[(Iterator[Product2[K,Any]],Int)]
    for((dep,depNum) <- dependencies.zipWithIndex)dep match {
      case oneToOneDependency:OneToOneDependency[Product2[K,Any]] @unchecked =>
        val dependencyPartition = split.narrowDeps(depNum).get.split
```

```
            // Read them from the parent
            val it = oneToOneDependency. rdd. iterator( dependencyPartition, context)
            rddIterators + = ( ( it, depNum) )

case shuffleDependency: ShuffleDependency[_,_,_] =>
//首先从 SparkEnv 获取 ShuffleManager
            //然后从 ShuffleDependency 中获取注册到 ShuffleManager 时
//所得到的 shuffleHandle。根据 shuffleHandle 和当前 Task 对应的分区 ID,
            //获取 ShuffleReader
//最后根据获取的 ShuffleReader,调用其 read 接口,读取 Shuffle 的 Map 输出
val it = SparkEnv. get. shuffleManager
              . getReader ( shuffleDependency. shuffleHandle, split. index, split. index + 1, context)
              . read( )
            rddIterators + = ( ( it, depNum) )
        }

        val map = createExternalMap( numRdds)
        for( ( it, depNum) <- rddIterators) {
          map. insertAll( it. map( pair => ( pair. _1, new CoGroupValue( pair. _2, depNum) ) ) )
        }
        context. taskMetrics( ). incMemoryBytesSpilled( map. memoryBytesSpilled)
        context. taskMetrics( ). incDiskBytesSpilled( map. diskBytesSpilled)
        context. internalMetricsToAccumulators(
InternalAccumulator. PEAK_EXECUTION_MEMORY). add( map. peakMemoryUsedBytes)
        new InterruptibleIterator( context,
          map. iterator. asInstanceOf[ Iterator[ ( K, Array[ Iterable[_] ] ) ] ] )
    }
```

从代码中可以看到，宽依赖的 RDD 的 compute 操作中，最终是通过 SparkEnv 中的 ShuffleManager 实例的 getReader 方法，获取数据的读取器的，然后再次调用读取器的 read 方法读取指定分区范围的 Shuffle 数据。

📖 注意，是宽依赖的 RDD，而非 ShuffleRDD。除了 ShuffleRDD 之外，还有其他 RDD 也可以是宽依赖，例如前面给出的 CoGroupedRDD。

目前支持的几种具体 Shuffle 实现机制在读取数据的处理上都是一样的，从源代码角度可以看到，当前继承了 ShuffleReader 这一数据读取器的接口的具体子类只有 BlockStoreShuffleReader，因此本章内容仅在此对各种 Shuffle 实现机制的数据读取进行解析，后续各实现机制中不再重复描述。

源代码解析的第一步是查看该类的描述信息，具体如下。

```
/**
  *通过从其他结点上请求读取 Shuffle 数据来接收并读取指定范围[起始分区,结束分
   区)——对应为左闭右开区间。


  */
```

从注释上可以看出，读取器负责上一 Stage 为下一 Stage 输出数据块的读取。从前面对 ShuffleReader 接口的解析可知，继承的具体子类需要实现真正的数据读取操作，即实现 read 方法。因此该方法是需要重点关注的源代码，一些关键的代码如下。

```
/** 为该 Reduce 任务读取并合并 key - values 值
  */
  override def read():Iterator[Product2[K,C]] = {
    //真正的数据 Iterator 读取是通过 ShuffleBlockFetcherIterator 来完成的
    val blockFetcherItr = new ShuffleBlockFetcherIterator(
    context,
    blockManager.shuffleClient,
    blockManager,
    //可以看到,当 ShuffleMapTask 完成后注册到 mapOutputTracker 的元数据信息
    //同样会通过 mapOutputTracker 来获取,在此同时还指定了获取的分区范围
    //通过该方法的返回值类型
mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId,startPartition,endPartition),
    // Note:we usegetSizeAsMb when no suffix is provided for backwards compatibility
    //默认读取时的数据大小限制为 48M,对应后续并行的读取,
//都是一种数据读取的控制策略,一方面可以避免目标机器占用过多带宽,
    //同时也可以启动并行机制,加快读取速度
     SparkEnv.get.conf.getSizeAsMb("spark.Reduce.maxSizeInFlight","48m") * 1024 *
     1024)

    // Wrap the streams for compression based on configuration
//在此针对前面获取的各个数据块唯一标识 ID 信息及其对应的输入流进行处理
 valw rappedStreams = blockFetcherItr.map {case(blockId,inputStream) =>
    blockManager.wrapForCompression(blockId,inputStream)
  }
…
  //对读取到的数据进行聚合处理
  val aggregatedIter:Iterator[Product2[K,C]] = if(dep.aggregator.isDefined) {
    //如果在 Map 端已经做了聚合的优化操作,则对读取到的聚合结果进行聚合,
    //注意此时的聚合操作与数据类型和 Map 端未做优化时是不同的
    if(dep.mapSideCombine) {
      // We are reading values that are already combined
```



```
            val combinedKeyValuesIterator = interruptibleIter. asInstanceOf[Iterator[(K,C)]]
            //针对 Map 端各分区对 Key 进行合并后的结果再次聚合,
            //Map 的合并可以大大减少网络传输的数据量
            dep. aggregator. get. combineCombinersByKey(combinedKeyValuesIterator,context)
        } else {
            // We don't know the value type,but also don't care -- the dependency * should *
            // have made sure its compatible w/ this aggregator,which will convert the value
            // type to the combined type C
            val keyValuesIterator = interruptibleIter. asInstanceOf[Iterator[(K,Nothing)]]
//针对未合并的 keyValues 的值进行聚合
dep. aggregator. get. combineValuesByKey(keyValuesIterator,context)
        }
    } else {
        require(!dep. mapSideCombine," Map - side combine without Aggregator specified!")
        interruptibleIter. asInstanceOf[Iterator[Product2[K,C]]]
    }

    //在基于 Sort 的 Shuffle 实现过程中,默认仅仅是基于 PartitionId 进行排序,
    //在分区的内部数据是没有排序的,因此添加了 keyOrdering 变量,
    //提供是否需要针对分区内的数据进行排序的标识信息
    // Sort the output if there is a sort ordering defined.
    dep. keyOrdering match {
        case Some(keyOrd:Ordering[K]) =>
            //为了减少内存的压力,避免 GC 开销,引入了外部排序器对数据进行排序
            //当内存不足以容纳排序的数据量时,会根据配置的 spark. shuffle. spill 属性
            //来决定是否需要 spill 到磁盘中,默认情况下会打开 spill 开关,
            //若不打开,在数据量比较大时会引发内存溢出问题(Out of Memory,OOM)
            //Create an ExternalSorter to sort the data. Note that if spark. shuffle. spill is disabled,
            // theExternalSorter won't spill to disk.
            val sorter =
                new ExternalSorter[K, C, C](context, ordering = Some(keyOrd), serializer = Some
(ser))
…
        case None =>
//不需要排序分区内部数据时直接返回
            aggregatedIter
    }
}
```

下面进一步解析数据读取的部分细节，首先是数据块获取，读取 ShuffleBlockFetcherIterator 类，在类的构造体中调用了 initialize 方法（构造体中的表达式会在构造实例时执行），该方法中会根据数据块所在位置（本地结点或远程结点）分别进行读取，其中的关键代码

如下。

```
private[this] def initialize():Unit = {
 …
//本地数据与远程数据的读取方式不同,因此先进行拆分,
    //注意拆分时会考虑一次获取的数据大小(拆分时会同时考虑并行数)封装请求,
    //最后会将剩余不足该大小的数据获取也封装为一个请求
    // Split local and remote blocks.
    val remoteRequests = splitLocalRemoteBlocks()
    // Add the remote requests into our queue in a random order
    //存入需要远程读取的数据块请求信息
    fetchRequests ++ = Utils.randomize(remoteRequests)


// Send out initial requests for blocks,up to ourmaxBytesInFlight
    //发送数据获取请求
    fetchUpToMaxBytes()
…
    //除了远程数据获取之外,下面是获取本地数据块的方法调用
    // Get Local Blocks
    fetchLocalBlocks()
    …
  }
```

与 Hadoop 一样，Spark 计算框架也是基于数据本地性，即移动计算而非移动数据的原则，因此在获取数据块时，也会考虑数据本地性，尽量从本地读取已有的数据块，然后再远程读取。

另外，数据块的本地性是通过 ShuffleBlockFetcherIterator 实例构建时所传入的位置信息来判断的，而该信息由 MapOutputTracker 实例的 getMapSizesByExecutorId 方法提供。可以参考该方法的返回值类型查看相关的位置信息，返回值类型为：Seq[(BlockManagerId,Seq[(BlockId,Long)])]，其中 BlockManagerId 是 BlockManager 的唯一标识信息，BlockId 是数据块的唯一信息，对应的 Seq[(BlockId,Long)])表示一组数据块标识 ID 及其数据块大小的元组信息。

最后简单分析一下如何设置分区内部的排序标识，当需要对分区内的数据进行排序时，会设置 RDD 中的宽依赖（ShuffleDependency）实例的 keyOrdering 变量。下面以基于排序的 OrderedRDDFunctions 提供的 sortByKey 方法为例给出解析，具体代码如下。

```
def sortByKey(ascending:Boolean = true,numPartitions:Int = self.partitions.length)
      :RDD[(K,V)] = self.withScope
  {
//注意,这里设置了该方法构建的 RDD 所使用的分区器
    // 根据 Range 而非 Hash 进行分区,对应的 Range 信息需要计算并将结果
   //反馈到 Driver 端,因此对应调用 RDD 中的 Action,即会触发一个 Job 的执行
```

```
val part = new RangePartitioner(numPartitions,self,ascending)
    //在构建 RDD 实例之后,设置 Key 的排序算法,即 Ordering 实例
    new ShuffledRDD[K,V,V](self,part)
      .setKeyOrdering(if(ascending)ordering else ordering.reverse)
  }
```

当需要对分区内部的数据进行排序时，构建 RDD 的同时会设置 Key 值的排序算法，结合前面的 read 方法中的第 52 行代码，当指定 Key 值的排序算法时，就会使用外部排序器对分区内的数据进行排序。

Section 7.3 基于 Hash 的 Shuffle

在 Spark 1.1 之前，Spark 中只实现了一种 Shuffle 方式，即基于 Hash 的 Shuffle。在 Spark 1.1 版本引入了基于 Sort 的 Shuffle 实现方式后，并且在 Spark 1.2 版本之后，默认的实现方式从基于 Hash 的 Shuffle 修改为基于 Sort 的 Shuffle 实现方式，即使用的 ShuffleManager 从默认的 hash 修改为 sort。

Spark 之所以一开始就提供基于 Hash 的 Shuffle 实现机制，其主要目的之一就是为了避免不必要的排序（这也是 Hadoop Map Reduce 被人所诟病的地方，将 Sort 作为固定步骤，导致了许多不必要的开销）。但基于 Hash 的 Shuffle 实现机制在处理超大规模数据集时，由于过程中会产生大量的文件，导致过度的磁盘 I/O 开销和内存开销，会极大地影响性能。

但在一些特定的应用场景下，采用基于 Hash 的实现 Shuffle 机制的性能会超过基于 Sort 的 Shuffle 实现机制。关于基于 Hash 与基于 Sort 的 Shuffle 实现机制的性能测试方面，可以参考 Spark 创始人之一 ReynoldXin 所给的测试：“sort – basedshuffle has lower memory usage and seems to outperformhash – based in almost allof our testing”。

相关数据可以参考 https://issues.apache.org/jira/browse/SPARK – 3280。

因此，在 Spark 1.2 版本中修改为默认基于 Sort 的 Shuffle 实现机制时，同时也给出了特定应用场景下回退的机制，具体可以参考 7.4 节基于 Sort 的 Shuffle 实现机制。

7.3.1 基于 Hash 的 Shuffle 内核

1. 基于 Hash 的 Shuffle 实现机制的内核框架

基于 Hash 的 Shuffle 实现，ShuffleManager 的具体实现子类为 HashShuffleManager，对应的具体实现机制如图 7–3 所示。

其中 HashShuffleManager 是 ShuffleManager 的基于 Hash 实现方式的具体实现子类。数据块的读写分别由 BlockStoreShuffleReader 与 HashShuffleWriter 实现；数据块的文件解析器则由具体子类 FileShuffleBlockResolver 实现；BaseShuffleHandle 是 ShuffleHandle 接口的基本实现，保存 Shuffle 注册的信息。

HashShuffleManager 继承自 ShuffleManager，对应实现了各个抽象接口，基于 Hash 的

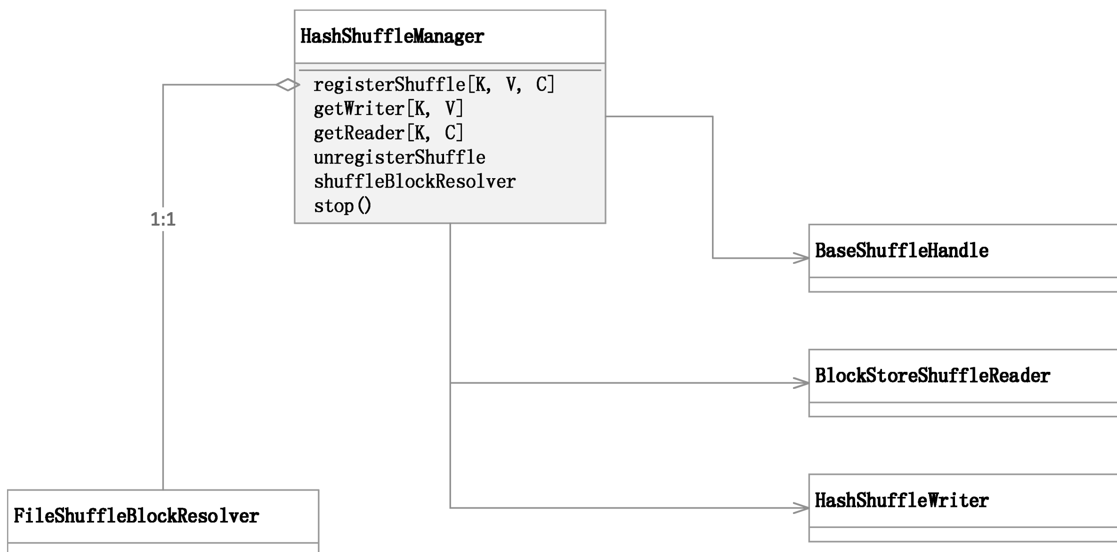


图 7-3　基于 Hash 的 Shuffle 实现机制的内核框架

Shuffle 内部使用的各组件的具体子类如下。

1）BaseShuffleHandle：携带了 Shuffle 最基本的元数据信息，包括 shuffleId、numMaps 和 dependency。

2）BlockStoreShuffleReader：负责写入的 Shuffle 数据块的读操作。

3）FileShuffleBlockResolver：负责管理为 Shuffle 任务分配基于磁盘的块数据的 Writer。每个 Shuffle 任务为每个 Reduce 分配一个文件。

4）HashShuffleWriter：负责 Shuffle 数据块的写操作。

在此与解析整个 Shuffle 过程一样，以 HashShuffleManager 类作为入口进行解析。

首先来看一下 HashShuffleManager 具体子类的注释，代码如下。

```
/**
  * 使用 Hash 的 ShuffleManager 具体实现子类,针对每个 Mapper 都会为各个 Reduce 分
  * 区构建一个输出文件(也可能是多个任务复用文件)。


  */
private[spark] class HashShuffleManager(conf:SparkConf) extends ShuffleManager with Logging {
…
```

2. 基于 Hash 的 Shuffle 实现方式一

为了避免 Hadoop 中基于 Sort 方式的 Shuffle 所带来的不必要的排序开销，Spark 在开始时采用了基于 Hash 的 Shuffle 方式。但这种方式存在很多缺陷，这些缺陷大部分是由于基于 Hash 的 Shuffle 实现过程中创建了太多的文件所造成的。在这种方式下，每个 Mapper 端的 Task 运行时都会为每个 Reduce 端的 Task 生成一个文件，具体如图 7-4 所示。

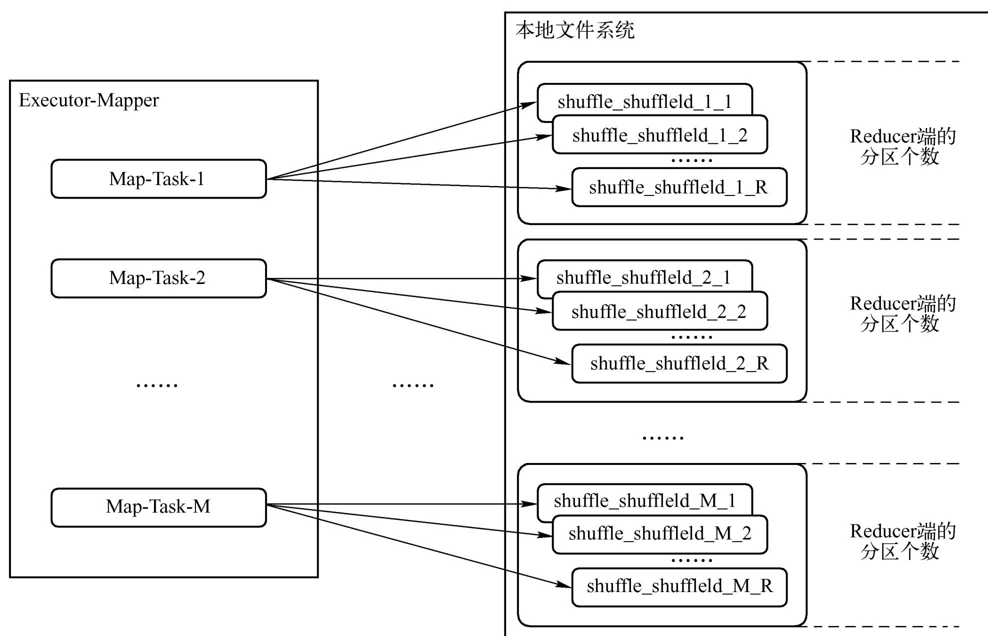


图 7-4　基于 Hash 的 Shuffle 实现方式一文件的输出细节图

图 7-4 中，Executor - Mapper 表示执行 Mapper 端的 Task 的工作点，可以分布到集群中的多台机器结点上，并且可以以不同的形式出现，比如以 Spark Standalone 部署模式中的 Executor 出现，也可以以 Spark On Yarn 部署模式中的容器形式出现，关键是它代表了实际执行 Mapper 端的 Tasks 的工作点的抽象概念。其中，M 表示 Mapper 端的 Task 的个数，R 表示 Reduce 端的 Task 的个数。

对应在右侧的本地文件系统是在该工作点上所生成的文件，其中 R 表示 Reduce 端的分区个数。生成的文件名格式为："shuffle_shuffleId_mapId_reduceId"，例如，其中的"shuffle_shuffleId_1_1"表示 mapId 为 1，同时 reduceId 也为 1。

在 Mapper 端，每个分区对应启动一个 Task，而每个 Task 会为每个 Reduer 端的 Task 生成一个文件，因此最终生成的文件个数为 M × R。

由于这种实现方式下，对应生成文件的个数仅仅与 Mapper 端和 Reduce 端各自的分区数有关，因此图 7-4 中将 Mapper 端的全部 M 个 Task 抽象到一个 Executor - Mapper 中，实际场景中通常是分布到集群中的各个工作点中。

生成的各个文件位于本地文件系统的指定目录中，该目录地址由配置属性"spark. local. dir"设置。

说明：分区数与 Task 数一个是静态的数据分块个数，一个是数据分块对应执行的动态任务个数的描述，因此在特定的描述个数的场景下，两者是一样的。

---

3. 基于 Hash 的 Shuffle 实现方式二

为了减少图 7-4 中所生成的文件个数，对基于 Hash 的 Shuffle 实现方式进行了优化，引入了文件合并的机制，该机制设置的开关为配置属性"spark. shuffle. consolidateFiles"。在引

入文件合并的机制之后，当设置配置属性为 true，即启动文件合并时，在 Mapper 端的输出文件会进行合并，在一定程度上可以大量减少文件的生成，降低不必要的开销。文件合并的实现方式可以参考图 7-5。

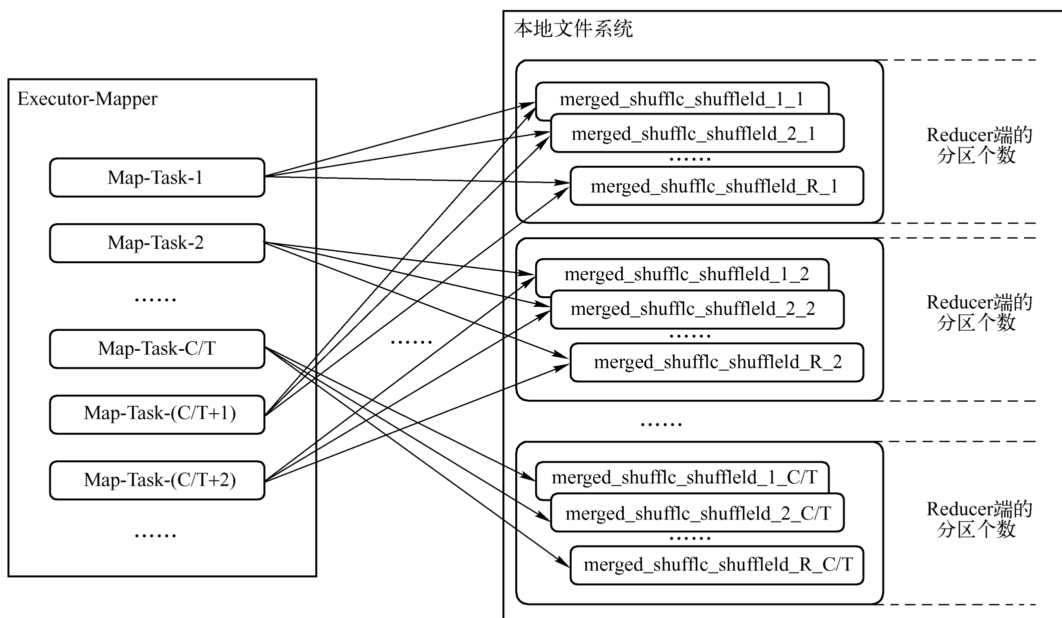


图 7-5　基于 Hash 的 Shuffle 的合并文件机制的输出细节图

图 7-5 中，Executor - Mapper 表示集群中分配的某个工作点，其中，C 表示在该工作点上所分配到的内核（Core）个数，T 表示在该工作点上为每个 Task 所分配的的内核个数。C/T 表示在该工作点上调度时最大的 Task 并行个数。

对应在右侧的本地文件系统是在该工作点上所生成的文件，其中 R 表示 Reduce 端的分区个数。生成的文件名格式为："merged_shuffle_shuffleId_bucketId_fileId"，其中的"merged_shuffle_shuffleId_1_1"表示 bucketId 为 1，同时 fileId 也为 1。

在 Mapper 端，Task 会复用文件组，由于最大并行个数为 C/T，因此文件组最多分配 C/T 个，当某个 Task 运行结束后会释放该文件组，之后调度的 Task 则复用前一个 Task 所释放的文件组，因此会复用同一个文件。最终在该工作点上生成的文件总数为 C/T × R，如果设工作点个数为 E，则总的文件数为 E × C/T × R。

4. 基于 Hash 的 Shuffle 机制的优缺点

1）优点如下。

- 可以省略不必要的排序开销。
- 避免了排序所需的内存开销。

2）缺点如下。

- 生成的文件数过多，会对文件系统造成压力。
- 大量小文件的随机读写会带来一定的磁盘开销。
- 数据块写入时所需的缓存空间也会随之增加，会对内存造成压力。

7.3.2 基于 Hash 的 Shuffle 写数据的源代码解析

1. 基于 Hash 的 Shuffle 实现方式一的源代码解析

下面继续针对 Spark 1.6 版本中的基于 Hash 的 Shuffle 实现在数据写方面的源代码进行解析。在基于 Hash 的 Shuffle 实现机制中，采用 HashShuffleWriter 作为数据写入器。在 HashShuffleWriter 中控制 Shuffle 写数据的关键代码如下。

```
private[spark] class HashShuffleWriter[K,V](
        shuffleBlockResolver:FileShuffleBlockResolver,
        handle:BaseShuffleHandle[K,V,_],
        mapId:Int,
        context:TaskContext)
    extends ShuffleWriter[K,V] with Logging {

    //控制每个 Writer 输出时的切片个数,对应分区个数
    private val dep = handle.dependency
    private val numOutputSplits = dep.partitioner.numPartitions

    …
    //获取数据读写的块管理器
    private val blockManager = SparkEnv.get.blockManager
    private val ser = Serializer.getSerializer(dep.serializer.getOrElse(null))

//从 FileShuffleBlockResolver 的 forMapTask 方法中获取指定的 shuffleId 对应的 mapId
//对应分区个数所构建的数据块写的 ShuffleWriterGroup 实例
    private val shuffle = shuffleBlockResolver.forMapTask(dep.shuffleId,mapId,numOutputSplits,ser,
        writeMetrics)

    /** Task 输出时一组记录的写入
    */
    override def write(records:Iterator[Product2[K,V]]):Unit = {
//判断在写时是否需要先聚合,即定义了 Map 端的 Combine 时,
//先对数据进行聚合再写入。否则直接返回需要写入的一批记录
        val iter = if(dep.aggregator.isDefined) {
            if(dep.mapSideCombine) {
                dep.aggregator.get.combineValuesByKey(records,context)
            } else {
                records
            }
        } else {
```

```
            require( ! dep. mapSideCombine, "Map - side combine without Aggregator specified!" )
            records
        }

        //根据分区器,获取每条记录对应的 bucketId(即所在的 Reduce 序号),
        //根据 bucketId 从 FileShuffleBlockResolver 构建的 ShuffleWriterGroup 中,获取
        //DiskBlockObjectWriter 实例,对应磁盘数据块的数据写入器
        for( elem <- iter) {
            val bucketId = dep. partitioner. getPartition( elem. _1 )
            shuffle. writers( bucketId). write( elem. _1 , elem. _2)
        }
    }
        …
}
```

当需要在 Map 端进行聚合时，使用的是聚合器(Aggregator)的 combineValuesByKey 方法，在该方法中使用 ExternalAppendOnlyMap 类对记录集进行处理，处理时如果内存不足会引发 Spill 操作。早期的实现会直接缓存到内存，在数据量比较大时容易引发内存泄漏。

在 HashShuffleManager 中，ShuffleBlockResolver 特质使用的具体子类为 FileShuffleBlockResolver，即指定了具体如何从一个逻辑 Shuffle 块标识信息来获取一个块数据，对应为下面代码第 7 行中的 forMapTask 方法，具体代码如下。

```
1.  /**
2.      * 针对给定的 Map Task,指定一个 ShuffleWriterGroup 实例,在数据块写入器成功
3.      * 关闭时会注册为完成状态
4.
5.
6.      */
7.  def forMapTask (shuffleId:Int, mapId:Int, numReduces:Int, serializer:Serializer,
8.            writeMetrics:ShuffleWriteMetrics):ShuffleWriterGroup = {
9.        new ShuffleWriterGroup {
10.           //在 FileShuffleBlockResolver 中维护着当前 Map Task 对应 shuffleId 标识的
11. //Shuffle 中,指定 numReduces 个数的 Reduce 的各个状态
12.           shuffleStates. putIfAbsent( shuffleId, new ShuffleState( numReduces) )
13.           private val shuffleState = shuffleStates( shuffleId)
14.
15. …
16.  //根据 Reduce 端的任务个数,构建元素类型为 DiskBlockObjectWriter 的数组,
17. // DiskBlockObjectWriter 负责具体数据的磁盘写入
18.    //原则上,Shuffle 的输出可以存放在各种提供存储机制的系统上,
19.  //但考虑到容错性等方面,目前的 Shuffle 实行机制都会写入到磁盘中
20.  val writers:Array[DiskBlockObjectWriter] = {
```

```
            //这里的逻辑 Bucket 的 Id 值即对应的 Reduce 的任务序号,或者说分区 ID
            Array.tabulate[DiskBlockObjectWriter](numReduces) { bucketId =>
//针对每个 Map 端分区的 Id 与 Bucket 的 Id 构建数据块的逻辑标识
              val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
              val blockFile = blockManager.diskBlockManager.getFile(blockId)
              val tmp = Utils.tempFileWith(blockFile)
              blockManager.getDiskWriter(blockId, tmp, serializerInstance, bufferSize, writeMetrics)
            }
          }
…
          //任务完成时回调的释放写入器方法
          override def releaseWriters(success: Boolean) {
            shuffleState.completedMapTasks.add(mapId)
          }
        }
      }
```

其中，ShuffleBlockId 实例构建的源代码如下。

```
case class ShuffleBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends BlockId {
  override def name: String = "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId
}
```

从 name 方法的重载上可以看出后续构建的文件与代码中的 mapId、reduceId 的关系。当然，同一个 Shuffle 的所有输出数据块都会带上 shuffleId 这个唯一标识的，因此从全局角度上看，逻辑数据块 name 不会重复（针对一些推测机制或失败重试机制之类的场景，逻辑 name 没有带上时间信息，因此缺少多次执行的输出区别，但在管理这些信息时会维护一个时间作为有效性判断）。

2. 基于 Hash 的 Shuffle 实现方式二的源代码解析

下面通过详细解析 FileShuffleBlockResolver 源代码来加深对文件合并机制的理解。

由于在 Spark 1.6 中，文件合并机制已经被删除，因此下面基于 Spark 1.5 版本的代码对文件合并机制的具体实现细节进行解析，以下代码位于 FileShuffleBlockResolver 类中。

合并机制的关键控制代码如下。

```
/**
   * 获取一个针对特定 Map Task 的 ShuffleWriterGroup

   * when the writers are closed successfully
   */
  def forMapTask(shuffleId: Int, mapId: Int, numBuckets: Int, serializer: Serializer,
        writeMetrics: ShuffleWriteMetrics): ShuffleWriterGroup = {
     new ShuffleWriterGroup {
```

```
9.  …
10.         val writers:Array[DiskBlockObjectWriter] = if(consolidateShuffleFiles){
11. //获取未使用的文件组
12. fileGroup = getUnusedFileGroup()
13.             Array.tabulate[DiskBlockObjectWriter](numBuckets){bucketId =>
14. val blockId = ShuffleBlockId(shuffleId,mapId,bucketId)
15. //注意获取磁盘写入器时,传入的第二个参数与未使用文件合并机制时的差异
16. //fileGroup(bucketId):构造器方式调用,对应到 apply 的方法调用
17.              blockManager.getDiskWriter(blockId, fileGroup(bucketId), serializerInstance, buffer-
             Size,
18.               writeMetrics)
19.             }
20.         } else {
21.             Array.tabulate[DiskBlockObjectWriter](numBuckets){bucketId =>
22. val blockId = ShuffleBlockId(shuffleId,mapId,bucketId)
23. //根据 ShuffleBlockId 信息获取文件名
24.               val blockFile = blockManager.diskBlockManager.getFile(blockId)
25.               val tmp = Utils.tempFileWith(blockFile)
26.               blockManager.getDiskWriter(blockId,tmp,serializerInstance,bufferSize,writeMetrics)
27.             }
28.         }
29. …
30.         writeMetrics.incShuffleWriteTime(System.nanoTime - openStartTime)
31.         override def releaseWriters(success:Boolean){
32. //带文件合并机制时,写入器在释放后的处理
33. //3 个关键信息:mapId、offsets 和 lengths
34.             if(consolidateShuffleFiles){
35.               if(success){
36.                 val offsets = writers.map(_.fileSegment().offset)
37.                 val lengths = writers.map(_.fileSegment().length)
38.                 fileGroup.recordMapOutput(mapId,offsets,lengths)
39.               }
40. //回收文件组,便于后续复用
41.               recycleFileGroup(fileGroup)
42.             } else {
43.               shuffleState.completedMapTasks.add(mapId)
44.             }
45.         }
46. }}}
```

其中，第 10 行中的 consolidateShuffleFiles 变量用于判断是否设置了文件合并机制，当设置 consolidateShuffleFiles 为 true 后，会继续调用 getUnusedFileGroup 方法，在该方法中会获取未使用的文件组，即重新分配或已经释放可以复用的文件组。

获取未使用的文件组(ShuffleFileGroup)的相关代码 getUnusedFileGroup 如下。

```
1.  private def getUnusedFileGroup():ShuffleFileGroup = {
2.  //获取已经构建但未使用的文件组,如果获取失败,则重新构建一个文件组
3.          val fileGroup = shuffleState.unusedFileGroups.poll()
4.          if(fileGroup != null)fileGroup else newFileGroup()
5.      }
6.  //重新构建一个文件组的源代码
7.      private def newFileGroup():ShuffleFileGroup = {
8.  //构建后会对文件编号进行递增,该文件编号最终用于生成的文件名中
9.          val fileId = shuffleState.nextFileId.getAndIncrement()
10.         val files = Array.tabulate[File](numBuckets) { bucketId =>
11. //最终的文件名,可以通过文件名的组成及取值细节,
12. //加深对实现细节在文件个数上的差异的理解
13.           val filename = physicalFileName(shuffleId,bucketId,fileId)
14.           blockManager.diskBlockManager.getFile(filename)
15.         }
16. //构建并添加到 shuffleState 中,便于后续复用
17.         val fileGroup = new ShuffleFileGroup(shuffleId,fileId,files)
18.         shuffleState.allFileGroups.add(fileGroup)
19.         fileGroup
20.     }
21.
```

其中，第 13 行代码对应为生成的文件名，即物理文件名，相关代码如下。

```
private def physicalFileName(shuffleId:Int,bucketId:Int,fileId:Int) = {
    "merged_shuffle_%d_%d_%d".format(shuffleId,bucketId,fileId)
  }
```

可以看到，与未使用文件合并时的基于 Hash 的 Shuffle 实现方式不同的是，在生成的文件名中没有对应的 mapId，取而代之的是与文件组相关的 fileId，而 fileId 则是多个 Mapper 端的 Task 所共用的，在此仅仅从生成的物理文件名中也可以看出文件合并的某些实现细节。

另外，对应生成的文件组既然是复用的，当一个 Mapper 端的 Task 执行结束后便会释放该文件组(ShuffleFileGroup)，之后继续调度时便会复用该文件组。相应的，调度到某个 Executor 工作点上同时运行的 Task 最大个数，就对应了最多分配的文件组个数。

而在 TaskSchedulerImpl 调度 Task 时，各个 Executor 工作点上 Task 调度控制的源代码说明了在各个 Executor 工作点上调度并行的 Task 数，具体代码如下。

```
private def resourceOfferSingleTaskSet(
    taskSet:TaskSetManager,
    maxLocality:TaskLocality,
    shuffledOffers:Seq[WorkerOffer],
```

```
        availableCpus:Array[Int],
        tasks:Seq[ArrayBuffer[TaskDescription]]):Boolean = {
var launchedTask = false
   for(i <- 0 until shuffledOffers.size){
val execId = shuffledOffers(i).executorId
val host = shuffledOffers(i).host
//判断当前 Executor 工作点上可用的内核个数是否满足 Task 所需的内核个数
//CPUS_PER_TASK:表示设置的每个 Task 所需的内核个数
if(availableCpus(i) > = CPUS_PER_TASK){
try{
for(task <- taskSet.resourceOffer(execId,host,maxLocality)){
…
           launchedTask = true
}
        } catch {
…
           }
        }
    }
returnlaunchedTask
}
```

其中，设置每个 Task 所需的内核个数的配置属性如下。

```
// CPUs to request per task
val CPUS_PER_TASK = conf.getInt("spark.task.cpus",1)
```

对于这些会影响 Executor 中并行执行的任务数的配置信息，在设置时需要多方面的考虑，包括内核个数与任务个数的合适比例，以及在内存模型中为任务分配内存的具体策略等。任务分配内存的具体策略可以参考 Spark 官方给出的具体设计文档，以及文档中各种设计方式的权衡等内容。

Section 7.4 基于 Sort 的 Shuffle

由于最初在 Spark 中为了避免不必要的排序开销而使用的基于 Hash 的 Shuffle 实现机制中存在一定的缺陷，在超大规模数据集的场景下会造成一定的性能瓶颈，对应的优化措施，即在 Spark 0.8.1 引入的 File Consolidation 也只是在一定程度上缓解了该问题，并没有彻底解决。因此在 SparkSpark 1.1.0 版本中，借鉴 Hadoop 的实现方式，引入了基于 Sort 的 Shuffle 实现机制。

基于 Sort 的 Shuffle 实现机制在 Spark 1.1.0 版本中被引进后，在 Spark 1.2 版本开始替换基于 Hash 的 Shuffle 实现机制，将基于 Sort 的 Shuffle 实现机制设置为默认方式。即将配置

属性 "spark. shuffle. manager" 从 hash 换成了 sort，对应 ShuffleManager 的实现类分别是 HashShuffleManager 和 SortShuffleManager。

在 Spark 1.1 版本中，Spark 借鉴了 Hadoop MapReduce 的基于 Sort 的 Shuffle 过程，不再为每个 Reduce 端的 Task 生成一个文件，而是根据分区 ID 进行排序，然后输出到单个数据文件中，并且同时生成对应的索引文件。在 Reduce 端获取数据时，会根据该索引文件从数据文件中获取它所需要的数据。因此在基于 Sort 的 Shuffle 过程中，可以避免基于 Hash 的 Shuffle 实现机制的缺陷，更有利于超大规模数据集的处理。

7.4.1 基于 Sort 的 Shuffle 内核

Sorted Based Shuffle，即基于 Sorted 的 Shuffle 实现机制，在该 Shuffle 过程中，Sorted 体现在输出的数据会根据目标的分区 Id（即带 Shuffle 过程的目标 RDD 中各个分区的 Id 值）进行排序，然后写入一个单独的 Map 端输出文件中。相应的，各个分区内部的数据并不会再根据 Key 值进行排序，除非调用带排序目的的方法，在方法中指定 Key 值的 Ordering 实例，才会在分区内部根据该 Ordering 实例对数据进行排序。当 Map 端的输出数据超过内存容纳大小时，会将各个排序结果 Spill 到磁盘上，最后再将这些 Spill 的文件合并到一个最终的文件中。在 Spark 的各种计算算子中到处都体现了一种惰性的理念，在此也是类似，在提升性能需要时，引入根据分区 Id 排序的设计，同时仅在指定分区内部排序的情况下才会进行全局排序。而相比之下，Hadoop 的 MapReduce 则带有一定的学术气息，中规中矩，严格设计 Shuffle 阶段中的各个步骤。

基于 Hash 的 Shuffle 实现，ShuffleManager 的具体实现子类为 HashShuffleManager，对应的具体实现机制如图 7-6 所示。

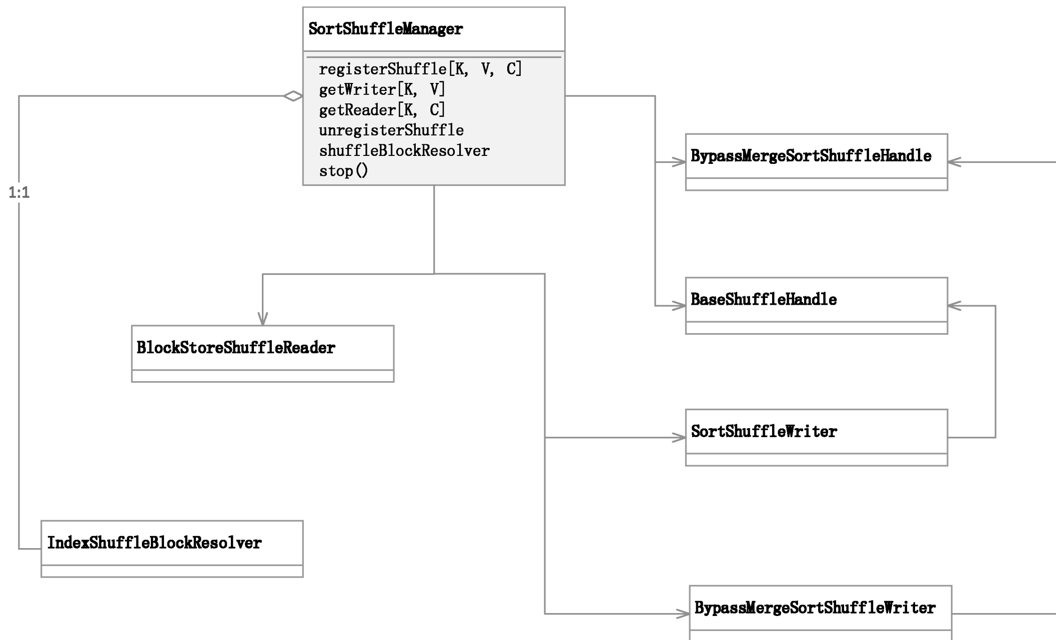


图 7-6 基于 Sorted 的 Shuffle 实现机制的框架类图

在图 7-6 中，各个不同的 ShuffleHandle 与不同的具体 Shuffle 写入器实现子类是一一对应的，可以认为是通过注册时生成的不同 ShuffleHandle 设置不同的 Shuffle 写入器实现子类。

从 ShuffleManager 注册的配置属性与具体实现子类的映射关系，即前面提及的在 SparkEnv 中实例化的代码，可以看出"sort"与"tungsten - sort"对应的具体实现子类都是"org. apache. spark. shuffle. sort. SortShuffleManager"。也就是当前基于 Sort 的 Shuffle 实现机制与使用 Tungsten 项目的 Shuffle 实现机制都是通过 SortShuffleManager 类来提供接口，两种实现机制的区别在于该类中使用了不同的 Shuffle 数据写入器。

SortShuffleManager 根据内部采用的不同实现细节，对应有两种不同的构建 Map 端文件输出的写方式（详细信息可以进一步参考 SortShuffleManager 的类注释），分别为序列化排序模式与反序列化排序模式。

1）序列化排序(Serialized sorting)模式：这种方式对应新引入的基于 Tungsten 项目的方式。

2）反序列化排序(Deserialized sorting)模式：这种方式对应除了前面这种方式之外的其他方式。

基于 Sort 的 Shuffle 实现机制采用的是反序列化排序(Deserialized sorting)模式，下面分析该实现机制下的数据写入器的实现细节。

在图 7-6 中可以看到，基于 Sort 的 Shuffle 实现机制，具体的写入器的选择与注册得到的 ShuffleHandle 类型有关，参考 SortShuffleManager 类的 registerShuffle 方法，相关代码如下。

```
    override def registerShuffle[K,V,C](
        shuffleId:Int,
        numMaps:Int,
        dependency:ShuffleDependency[K,V,C]):ShuffleHandle = {
      //通过 shouldBypassMergeSort 方法判断是否满足回退到 Hash 风格的 Shuffle 条件
      if(SortShuffleWriter. shouldBypassMergeSort(SparkEnv. get. conf,dependency)){
        //如果当前的分区个数小于设置的配置属性：
        // spark. shuffle. sort. bypassMergeThreshold,同时不需要在 Map 对数据进行聚合,
        //此时可以直接写文件,并在最后将文件合并
//
        //
        // This avoids doing serialization anddeserialization twice
        // to merge together the spilled files,which would happen with the normal code path.
        // The downside is having multiple files open at a time and thus more memory
//allocated to buffers.
        new BypassMergeSortShuffleHandle [K,V](
          shuffleId,numMaps,dependency. asInstanceOf[ShuffleDependency[K,V,V]])
      } else if(SortShuffleManager. canUseSerializedShuffle(dependency)){
       …
      } else {
        // Otherwise,buffer map outputs in adeserialized form:
```

```
            new BaseShuffleHandle(shuffleId,numMaps,dependency)
        }
    }
```

7.4.2 基于 Sort 的 Shuffle 写数据的源代码解析

基于 Sort 的 Shuffle 实现机制中相关的 ShuffleHandle 包含 BypassMergeSortShuffleHandle 与 BaseShuffleHandle。对应这两种 ShuffleHandle 及其相关的 Shuffle 数据写入器类型的相关代码可以参考 SortShuffleManager 类的 getWriter 方法，关键代码如下。

```
/** Get a writer for a given partition. Called on executors by map tasks. */
override def getWriter[K,V](
    handle:ShuffleHandle,
    mapId:Int,
    context:TaskContext):ShuffleWriter[K,V] = {
  numMapsForShuffle.putIfAbsent(
    handle.shuffleId,handle.asInstanceOf[BaseShuffleHandle[_,_,_]].numMaps)
  val env = SparkEnv.get
  //通过对 ShuffleHandle 类型的模式匹配,构建具体的数据写入器
  handle match {
    ...
     case bypassMergeSortHandle: BypassMergeSortShuffleHandle[K @ unchecked, V @ unchecked] =>
      new BypassMergeSortShuffleWriter(
        env.blockManager,
        shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],
        bypassMergeSortHandle,
        mapId,
        context,
        env.conf)
    case other:BaseShuffleHandle[K @ unchecked,V @ unchecked,_] =>
      new SortShuffleWriter(shuffleBlockResolver,other,mapId,context)
  }
}
```

在对应构建的两种数据写入器类 BypassMergeSortShuffleWriter 与 SortShuffleWriter 中，都是通过变量 shuffleBlockResolver 对逻辑数据块和物理数据块的映射进行解析，而该变量使用的是与基于 Hash 的 Shuffle 实现机制不同的解析类，即当前使用 IndexShuffleBlockResolver。

下面开始解析这两种写数据块方式的源代码实现。

1. BypassMergeSortShuffleWriter 写数据的源代码解析

BypassMergeSortShuffleWriter 类实现了带 Hash 风格的基于 Sort 的 Shuffle 机制，为每个

Reduce 端的任务构建一个输出文件，将输入的每条记录分别写入各自对应的文件中，并在最后将这些基于各个分区的文件合并成一个输出文件。

从前面提到的实例中可以知道，在 Rueducer 端任务数比较少的情况下，基于 Hash 的 Shuffle 实现机制明显比基于 Sort 的 Shuffle 实现机制要快，因此基于 Sort 的 Shuffle 实现机制提供了一个 fallback 方案，当 Rueducer 端任务数少于配置属性“spark. shuffle. sort. bypassMergeThreshold”设置的个数时，使用带 Hash 风格的 fallback 计划，由 BypassMergeSortShuffleWriter 具体实现。

使用该写入器的条件如下。

1）不能指定 Ordering，从前面数据读取器的解析可以知道，当指定 Ordering 时，会对分区内部的数据进行排序。因此，对应的 BypassMergeSortShuffleWriter 写入器避免了排序开销。

2）不能指定聚合器（Aggregator）。

3）分区个数小于“spark. shuffle. sort. bypassMergeThreshold”配置属性指定的个数。

和其他 ShuffleWriter 的具体子类一样，BypassMergeSortShuffleWriter 写数据的具体实现位于实现的 write 方法中，关键代码如下。

```
public void write(Iterator<Product2<K,V>> records) throws IOException {
//为每个 Reduce 端的分区打开的 DiskBlockObjectWriter 存放于 partitionWriters,
    //需要根据具体 Reduce 端的分区个数进行构建
    assert(partitionWriters == null);
    if(!records.hasNext()) {
      partitionLengths = new long[numPartitions];
      //初始化索引文件的内容,此时对应各个分区的数据量或偏移量需要在后续
      //获取分区的真实数据量时重写
      shuffleBlockResolver.writeIndexFileAndCommit(shuffleId,mapId,partitionLengths,null);
      //下面代码的调用形式是对应在 Java 类中调用 Scala 提供的 object 中的
      //apply 方法的形式,这是由编译器编译 Scala 中的 object 得到的结果来决定的
       mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partition-
       Lengths);
      return;
    }
    final SerializerInstance serInstance = serializer.newInstance();
    final longopenStartTime = System.nanoTime();
    //对应每个分区各配置一个磁盘写入器 DiskBlockObjectWriter
    partitionWriters = new DiskBlockObjectWriter[numPartitions];
    //注意,在该写入方式下,会同时打开 numPartitions 个 DiskBlockObjectWriter,
    //因此对应的分区数不应设置得过大,避免带来过重的内存开销。
    //目前对应 DiskBlockObjectWriter 的缓存大小默认配置为 32K,
    //比早先的 100 K 降低了很多,
    //但也说明不适合同时打开太多的 DiskBlockObjectWriter 实例
    for(int i = 0; i < numPartitions; i++) {
```

```
25.         final Tuple2 < TempShuffleBlockId, File > tempShuffleBlockIdPlusFile =
26. blockManager. diskBlockManager( ). createTempShuffleBlock( );
27.         final File file = tempShuffleBlockIdPlusFile. _2( );
28.         finalBlockId blockId = tempShuffleBlockIdPlusFile. _1( );
29.         partitionWriters[ i] =
30.             blockManager. getDiskWriter ( blockId, file, serInstance, fileBufferSize, writeMetrics )
              . open( );
31.       }
32.       …
33. //读取每条记录,并根据分区器将该记录交由分区对应的 DiskBlockObjectWriter,
34.     //写入各自对应的临时文件中
35.     while( records. hasNext( ) ) {
36.         final Product2 < K, V > record = records. next( );
37.         final K key = record. _1( );
38.         partitionWriters[ partitioner. getPartition( key) ]. write( key, record. _2( ) );
39.       }
40.
41.       …
42. //获取最终合并后的文件名,对应格式为:
43.     //"shuffle_" + shuffleId + "_" + mapId + "_" + reduceId + ". index",并且其中的 reduceId
44.     //为 0,对应的含义就是该文件包含所有为 Reduce 端输出的数据
45.     File output = shuffleBlockResolver. getDataFile( shuffleId, mapId);
46.     Filetmp = Utils. tempFileWith( output);
47.     //在此合并前面生成的各个中间临时文件,并获取各个分区对应的数据量,
48.     //由数据量可以得到对应的偏移量
49.     partitionLengths = writePartitionedFile( tmp);
50.     //主要是根据前面获取的数据量,重写 Index 文件中的偏移量信息
51.     shuffleBlockResolver. writeIndexFileAndCommit( shuffleId, mapId, partitionLengths, tmp);
52.     //封装并返回任务结果
53.      mapStatus = MapStatus $ . MODULE $ . apply ( blockManager. shuffleServerId ( ) , partition-
     Lengths);
54.   }
```

其中第 26 行代码，描述了各个分区所生成的中间临时文件的格式与对应的 BlockId，具体代码如下。

```
  /** Produces a unique block id and File suitable for storing shuffled intermediate results. */
/** 中间临时文件名的格式由前缀 temp_shuffle_与 randomUUID 组成,可以唯一标识一个
BlockId */
  def createTempShuffleBlock( ): (TempShuffleBlockId, File) = {
    var blockId = new TempShuffleBlockId( UUID. randomUUID( ) )
    while( getFile( blockId). exists( ) ) {
```




```
            blockId = new TempShuffleBlockId(UUID.randomUUID())
        }
        (blockId,getFile(blockId))
    }
```

从上面的分析可以知道，每个 Map 端的任务最终会生成两个文件：数据（Data）文件和索引（Index）文件。

另外在使用 DiskBlockObjectWriter 写记录时，是以 32 条记录批次写入的，不会占用太大的内存。但由于不能指定聚合器(Aggregator)，写数据时也是直接写入记录，因此对应后续的网络 I/O 的开销也会很大。

2. SortShuffleWriter 写数据的源代码解析

前面 BypassMergeSortShuffleWriter 的写数据是在 Reduce 端的分区个数较少的情况下提供的一种优化方式，但当数据集规模非常大，使用该写数据方式不合适时，就需要使用 SortShuffleWriter 来写数据块。

和其他 ShuffleWriter 的具体子类一样，SortShuffleWriter 写数据的具体实现位于实现的 write 方法中，关键代码如下。

```
  override def write(records:Iterator[Product2[K,V]]):Unit = {
    //当需要在 Map 端进行聚合操作时,此时会将指定的聚合器(Aggregator)
//与 Key 值的 Ordering 传入到外部排序器 ExternalSorter 中
    sorter = if(dep.mapSideCombine) {
      require(dep.aggregator.isDefined,"Map-side combine without Aggregator specified!")
      new ExternalSorter[K,V,C](
        context,dep.aggregator,Some(dep.partitioner),dep.keyOrdering,dep.serializer)
    } else {
      //没有指定 Map 端使用聚合时,传入 ExternalSorter 的聚合器(Aggregator)
    //与 Key 值的 Ordering 都设为 None,即不需要传入,对应在 Reduce 端读取数据
      //时才根据聚合器分区数据进行聚合,并根据是否设置 Ordering 而选择是否对
    //分区数据进行排序
      // In this case we pass neither an aggregator nor an ordering to the sorter,because we
      // don't care whether the keys get sorted in each partition; that will be done on the
      // reduce side if the operation being run issortByKey.
      new ExternalSorter[K,V,V](
        context,aggregator = None,Some(dep.partitioner),ordering = None,dep.serializer)
    }
//将写入的记录集全部放入外部排序器
    sorter.insertAll(records)

    // Don't bother including the time to open the merged output file in the shuffle write time,
    // because it just opens a single file,so is typically too fast to measure accurately
    //(see SPARK-3570).
    //和 BypassMergeSortShuffleWriter 一样,获取输出文件名和 BlockId
```

```
26.        val output = shuffleBlockResolver.getDataFile(dep.shuffleId,mapId)
27.        val tmp = Utils.tempFileWith(output)
28.        val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP_RE-
           DUCE_ID)
29.        //将分区数据写入文件,返回各个分区对应的数据量
30.        val partitionLengths = sorter.writePartitionedFile(blockId,tmp)
31.
32.        //和 BypassMergeSortShuffleWriter 一样,更新索引文件的偏移量信息
33.        shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId,mapId,partitionLengths,tmp)
34.        mapStatus = MapStatus(blockManager.shuffleServerId,partitionLengths)
35.    }
```

这种基于 Sort 的 Shuffle 实现机制中引入了外部排序器(ExternalSorter)，ExternalSorter 继承了 Spillable，因此内存使用在达到一定阈值时，会 Spill 到磁盘，可以减少内存带来的开销。

通过查看外部排序器(ExternalSorter)的 insertAll 方法，对应调用在第 20 行，该方法内部在处理完（包含聚合和非聚合两种方式）每一条记录时，都会检查是否需要 Spill。如果内部各种细节比较多，这里以 Spill 条件判断为主线，简单描述与条件相关的代码。具体判断是否需要 Spill 的相关代码，可以参考 Spillable 类中的 maybeSpill 方法（该方法简单的调用流程为：ExternalSorter #insterAll→ExternalSorter #maybeSpillCollection→Spillable#maybeSpill)，关键代码如下。

```
protected def maybeSpill(collection:C,currentMemory:Long):Boolean = {
   //判断是否需要 Spill
   var shouldSpill = false
     // 1. 检查当前记录数是否是 32 的倍数,即对小批量的记录集进行 Spill
// 2. 同时,当前需要的内存大小是否达到或超过了当前分配的内存阈值
     if(elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) {
       // Claim up to double our current memory from the shuffle memory pool
       val amountToRequest = 2 * currentMemory - myMemoryThreshold
//实际上会先申请内存,然后再次判断,再决定是否 Spill
val granted =
         taskMemoryManager.acquireExecutionMemory(amountToRequest, MemoryMode.ON_
         HEAP,null)
       myMemoryThreshold += granted
       // If we were granted too little memory to grow further(eithertryToAcquire returned 0,
       // or we already had more memory thanmyMemoryThreshold),spill the current collection
       shouldSpill = currentMemory >= myMemoryThreshold
     }
     //当满足下列条件之一时,需要 Spill,条件如下。
     // 1. 当前面判断结果为 true 时
     // 2. 从上次 Spill 之后所读取的记录数超过配置的阈值时
```

```
        //配置属性为:"spark.shuffle.spill.numElementsForceSpillThreshold"
        shouldSpill = shouldSpill || _elementsRead > numElementsForceSpillThreshold
        // Actually spill
        if(shouldSpill){
          _spillCount += 1
          logSpillage(currentMemory)
          spill(collection)
          _elementsRead = 0
          _memoryBytesSpilled += currentMemory
          releaseMemory()
        }
        shouldSpill
    }
```

对于外部排序器（ExternalSorter），除了 insertAll 方法外，它的 writePartitionedFile 方法也非常重要。该方法的代码如下。

```
def writePartitionedFile(
      blockId: BlockId,
      outputFile: File): Array[Long] = {
```

其中，BlockId 是数据块的逻辑位置，File 参数则是对应逻辑位置的物理存储位置。这两个参数值的获取方法和使用 BypassMergeSortShuffleHandle 及其对应 ShuffleWriter 是一样的。

在该方法中，有一个容易混淆的地方，与 Shuffle 的度量（Metric）信息有关，相应的代码如下。

```
1. context.taskMetrics().incMemoryBytesSpilled(memoryBytesSpilled)
2. context.taskMetrics().incDiskBytesSpilled(diskBytesSpilled)
```

其中，第 1 行对应修改了 Spilled 的数据在内存中的字节大小，第 2 行则对应修改了 Spilled 的数据在磁盘中的字节大小。在内存中时，数据是反序列化形式存放的，而存储到磁盘（默认会序列化）时，会对数据进行序列化。反序列化后的数据会远远大于序列化后的数据（也可以通过 UI 界面查看这两个度量信息的大小差异来确认，具体差异的大小和数据与选择的序列化器有关，有兴趣的读者可以参考各序列器间的性能等比较文档）。

从这一点也可以看出，如果在内存中使用反序列化的数据，会大大增加内存的开销（也意味着增加 GC 负载），并且反序列化也会增加 CPU 的开销，因此引入了利用 Tungsten 项目的基于 Tungsten Sort 的 Shuffle 实现机制，Tungsten 项目的优化主要有 3 个方面，这里从避免反序列化的数据量会极大地消耗内存这点考虑，主要是借助 Tungsten 项目的内存管理模型，可以直接处理序列化的数据；同时在 CPU 开销方面，直接处理序列化数据，可以避免数据反序列化的这部分处理开销。

Section

7.5 基于 Tungsten Sort 的 Shuffle

基于 Tungsten Sort 的 Shuffle 实现机制主要是借助 Tungsten 项目所做的优化来高效处理 Shuffle。

Spark 提供了配置属性，用于选择具体的 Shuffle 实现机制，但需要说明的是，虽然默认情况下，Spark 默认开启的是基于 Sort 的 Shuffle 实现机制（对应 spark. shuffle. manager 的默认值），但实际上，参考 Shuffle 的框架内核部分可知，基于 Sort 的 Shuffle 实现机制与基于 Tungsten Sort 的 Shuffle 实现机制都是使用 SortShuffleManager，而内部使用的具体实现机制是通过提供的两个方法进行判断的。对应非基于 Tungsten Sort 时，通过 SortShuffleWriter. shouldBypassMergeSort 方法判断是否需要回退到 Hash 风格的 Shuffle 实现机制，当该方法返回的条件不满足时，则通过 SortShuffleManager. canUseSerializedShuffle 方法判断是否需要采用基于 Tungsten Sort 的 Shuffle 实现机制，而在这两个方法的返回都为 false，即都不满足对应的条件时，会自动采用常规意义上的基于 Sort 的 Shuffle 实现机制。

因此，当设置了 spark. shuffle. manager = tungsten – sort 时，也不能保证就一定采用基于 Tungsten Sort 的 Shuffle 实现机制。有兴趣的读者可以参考 Spark 1.5 及之前的注册方法的实现，该实现中 SortShuffleManager 的注册方法仅构建了 BaseShuffleHandle 实例，同时对应的 getWriter 中也对应只构建了 BaseShuffleHandle 实例。

7.5.1 基于 Tungsten Sort 的 Shuffle 内核

基于 Tungsten Sort 的 Shuffle 实现机制的入口点仍然是 SortShuffleManager 类，与同样在 SortShuffleManager 类控制下的其他两种实现机制不同的是，基于 Tungsten Sort 的 Shuffle 实现机制使用的 ShuffleHandle 与 ShuffleWriter 分别为 SerializedShuffleHandle 与 UnsafeShuffleWriter。因此对应的具体实现机制可以用图 7-7 来表示。

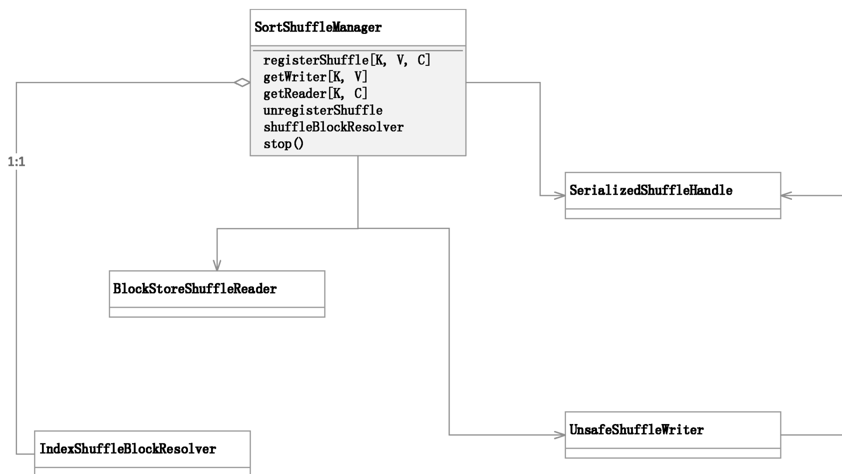


图 7-7 基于 TungstenSort 的 Shuffle 实现机制的框架类图

在基于 Sort 的 Shuffle 内核一节中已经提到过 SortShuffleManager 根据内部采用的不同实现细节，分别给出两种排序模式，而基于 Tungsten Sort 的 Shuffle 实现机制对应的就是序列化排序模式。

从图 7-7 中可以看到，基于 Sort 的 Shuffle 实现机制，具体的写入器的选择与注册得到的 ShuffleHandle 类型有关，参考 SortShuffleManager 类的 registerShuffle 方法，相关代码如下。

```
1.  /**
2.      * Register a shuffle with the manager and obtain a handle for it to pass to tasks.
3.   * 向 ShuffleManager 注册一个 Shuffle 并获取一个 ShuffleHandle,
4.      * 然后传入到 Tasks 中。在 Map 端的 Tasks 通过该 ShuffleHandle 从 ShuffleManager
5.   * 获取数据写入器 ShuffleWriter,对应 Reduce 端的 Tasks 通过该 ShuffleHandle
6.   * 从 ShuffleManager 获取数据读取器 ShuffleReader
7.      */
8.    override def registerShuffle[K,V,C](
9.        shuffleId:Int,
10.       numMaps:Int,
11.       dependency:ShuffleDependency[K,V,C]):ShuffleHandle = {
12.       if(SortShuffleWriter.shouldBypassMergeSort(SparkEnv.get.conf,dependency)){
13. …
14. //
15. } else if(SortShuffleManager.canUseSerializedShuffle(dependency)){
16.       // Otherwise,try to buffer map outputs in a serialized form,since this is more efficient:
17.       //当可以使用序列化模式时,可以直接以序列化的个数输出数据,这可以减少内
18. //存开销和序列化、反序列化方面的 CPU 开销等,因此会更加高效
19.       new SerializedShuffleHandle[K,V](
20.         shuffleId,numMaps,dependency.asInstanceOf[ShuffleDependency[K,V,V]])
21.     } else {
22. …
23.     }
24.   }
```

在第 15 行代码处，会判断是否满足序列化模式的条件，如果满足，则使用基于 TungstenSort 的 Shuffle 实现机制，对应在代码中表现为使用类型为 SerializedShuffleHandle 的 ShuffleHandle。上述代码进一步说明了在将“spark.shuffle.manager”设置为 sort 时，内部会自动选择具体的实现机制。对应代码的先后顺序就是选择的先后顺序。

对应的序列化排序（Serialized sorting）模式需要满足的条件如下。

1）Shuffle 依赖中不带聚合操作或没有对输出进行排序的要求。

2）Shuffle 的序列化器支持序列化值的重定位（当前仅仅支持 KryoSerializer 及 Spark SQL 子框架自定义的序列化器）。

3）Shuffle 过程中的输出分区个数少于 16777216 个。

实际上，使用过程中还有其他一些限制，比如由于使用 Page 形式的内存管理模型后，内部单条记录的长度不能超过 128 MB（具体内存模型可以参考 PackedRecordPointer 类）。另

外分区个数的限制也是该内存模型导致的（同样参考 PackedRecordPointer 类）。

所以，目前使用基于 TungstenSort 的 Shuffle 实现机制的条件还是比较苛刻的。

7.5.2 基于 Tungsten Sort 的 Shuffle 写数据的源代码解析



对应这种 SerializedShuffleHandle 及其相关的 Shuffle 数据写入器类型的相关代码可以参考 SortShuffleManager 类的 getWriter 方法，关键代码如下。

```
1.     /** 为指定的分区提供一个数据写入器。该方法在 Map 端的 Tasks 中调用 */
2.     /** Get a writer for a given partition.  Called on executors by map tasks.  */
3.     override def getWriter[K,V](
4.         handle:ShuffleHandle,
5.         mapId:Int,
6.         context:TaskContext):ShuffleWriter[K,V] = {
7.       numMapsForShuffle.putIfAbsent(
8.         handle.shuffleId,handle.asInstanceOf[BaseShuffleHandle[_,_,_]].numMaps)
9.       val env = SparkEnv.get
10.      handle match {
11. //SerializedShuffleHandle 对应的写入器为 UnsafeShuffleWriter
12.        //使用的数据块逻辑与物理映射关系仍然为 IndexShuffleBlockResolver,对应
13.        //SortShuffleManager 中的变量,因此相同
14.        case unsafeShuffleHandle:SerializedShuffleHandle[K @unchecked,V @unchecked] =>
15. new UnsafeShuffleWriter(
16.            env.blockManager,
17.            shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],
18.            context.taskMemoryManager(),
19.            unsafeShuffleHandle,
20.            mapId,
21.            context,
22.            env.conf)
23.      }
24.    }
```

在数据写入器类 UnsafeShuffleWriter 中，使用的仍然是 SortShuffleManager 实例中的变量 shuffleBlockResolver 来对逻辑数据块与物理数据块的映射进行解析，而该变量使用的是与基于 Hash 的 Shuffle 实现机制不同的解析类，即当前使用 IndexShuffleBlockResolver。

在第 18 行代码中，可以看到 UnsafeShuffleWriter 构建时传入了一个与其他两种基于 Sorted 的 Shuffle 实现机制不同的参数：context.taskMemoryManager()，在此构建了一个 TaskMemoryManager 实例并传入 UnsafeShuffleWriter，TaskMemoryManager 与 Task 是一对一的关系，负责管理分配给 Task 的内存。

下面开始解析写数据块的 UnsafeShuffleWriter 类的源代码实现。首先查看其 write 的方法，代码如下。

```
1.      @ Override
2.      public void write(scala.collection.Iterator<Product2<K,V>> records) throws IOException {
3.    …
4.        try {
5.          //对输入的记录集 records,循环将每条记录插入到外部排序器。
6.          while(records.hasNext()) {
7.            insertRecordIntoSorter(records.next());
8.          }
9.          //生成最终的两个结果文件,和 Sorted Based Shuffle 的实现机制一样,
10.   //每个 Map 端的任务对应生成一个数据(Data)文件和对应的索引(Index)文件。
11.         closeAndWriteOutput();
12.         success = true;
13.       } finally {
14.         if(sorter != null) {
15.           try {
16.             //释放排序过程中使用的资源
17.             sorter.cleanupResources();
18.           } catch(Exception e) {
19.           …
20.         }
21.       }
22.     }
23.   }
```

写过程中的关键步骤只有 3 步。

1）第 5 ~7 行，将每条记录插入外部排序器。

2）第 11 行，写数据文件与所有文件，在写的过程中，会先合并外部排序器在插入过程中生成的 Spill 中间文件。

3）第 17 行，最后释放外部排序器的资源。

首先查看将每条记录插入外部排序器（ShuffleExternalSorter）时所使用的 insertRecordIntoSorter 方法，其关键代码如下。

```
void insertRecordIntoSorter(Product2<K,V> record) throwsIOException {
    assert(sorter != null);
//对于多次访问的 Key 值,使用局部变量,可以避免多次函数调用
//有兴趣的读者可以跟踪一下 spark 的 issues 相关内容,对于元组在
    //模式匹配等方面优化的一些细节处理
    final K key = record._1();
    final intpartitionId = partitioner.getPartition(key);
//先复位存放每条记录的缓冲区
    //内部使用 ByteArrayOutputStream 存放每条记录,容量为 1MB
    serBuffer.reset();
    //进一步使用序列化器从 serBuffer 缓冲区构建序列化输出流,将记录写入到缓冲区
    serOutputStream.writeKey(key,OBJECT_CLASS_TAG);
```

```
        serOutputStream.writeValue(record._2(),OBJECT_CLASS_TAG);
        serOutputStream.flush();

        final int serializedRecordSize = serBuffer.size();
        assert(serializedRecordSize > 0);

        //将记录插入到外部排序器中,serBuffer 是一个字节数组
        //内部数据存放的偏移量为 Platform.BYTE_ARRAY_OFFSET
        sorter.insertRecord(
          serBuffer.getBuf(),Platform.BYTE_ARRAY_OFFSET,serializedRecordSize,partitionId);
    }
```

下面继续查看第二步写数据文件与索引文件的 closeAndWriteOutput 方法，其关键代码如下。

```
    void closeAndWriteOutput()throws IOException {
        …
        //设为 null,用于 GC 垃圾回收
        serBuffer = null;
        serOutputStream = null;
    //关闭外部排序器并获取全部 Spill 信息
        finalSpillInfo[] spills = sorter.closeAndGetSpills();
        sorter = null;
        final long[]partitionLengths;
        //通过块解析器获取输出文件名
        final File output = shuffleBlockResolver.getDataFile(shuffleId,mapId);
        //在后续合并 Spill 文件时先使用临时文件名,最终再重命名为真正的输出文件名。
        //即在 writeIndexFileAndCommit 方法中会重复通过块解析器获取输出文件名
        final Filetmp = Utils.tempFileWith(output);
        try {
          partitionLengths = mergeSpills(spills,tmp);
        } finally {
          //清理中间生成的 Spill 文件
           …
        }
        //将合并 Spill 后获取的分区及其数据量信息写入索引文件,
        //并将临时数据文件重命名为真正的数据文件名
        shuffleBlockResolver.writeIndexFileAndCommit(shuffleId,mapId,partitionLengths,tmp);
         mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(),partitionLengths);
    }
```



closeAndWriteOutput 方法主要有以下几步。

1）触发外部排序器，获取 Spill 信息。

2）合并中间的 Spill 文件，生成数据文件，并返回各个分区对应的数据量信息。

3）根据各个分区的数据量信息生成数据文件对应的索引文件。

由于 writeIndexFileAndCommit 方法和前面 Sorted Based Shuffle 机制的实现一样，在此仅分析过程中不同的 Spill 文件合并步骤，即 mergeSpills 方法的具体实现。

mergeSpills 方法的关键代码如下。

```
/**
  * Merge zero or more spill files together, choosing the fastest merging strategy based on
  * the number of spills and the IO compression codec.
  *
  * @return the partition lengths in the merged file.
  *
  * 合并 0 个或多个 spill 的中间文件,基于 Spills 的个数以及 IO 压缩码选择最快
  * 速的合并策略。返回包含合并文件中各个分区的数据长度的数组。
  */
private long[] mergeSpills(SpillInfo[] spills, File outputFile) throws IOException {
  //获取 Shuffle 的压缩配置信息
  final boolean compressionEnabled = sparkConf.getBoolean("spark.shuffle.compress", true);
  final CompressionCodec compressionCodec = CompressionCodec$.MODULE$.createCodec(sparkConf);
  //获取是否启动 unsafe 的快速合并
  final boolean fastMergeEnabled =
    sparkConf.getBoolean("spark.shuffle.unsafe.fastMergeEnabled", true);
  //没有压缩或者当压缩码支持序列化流合并时,支持快速合并
  final boolean fastMergeIsSupported = ! compressionEnabled ||
CompressionCodec$.MODULE$.supportsConcatenationOfSerializedStreams(compressionCodec);
  try {
    if(spills.length == 0) {
      //没有中间的 spills 文件时,创建一个空文件,并返回包含分区数据长度的
      //空数组。后续读取时会过滤掉空文件
      new FileOutputStream(outputFile).close(); // Create an empty file
      return new long[partitioner.numPartitions()];
    } else if(spills.length == 1) {
      //最后一个 spills 文件已经更新 metrics 信息,因此不需要重复更新
      //直接重命名 spills 的中间临时文件为目标输出的数据文件
      //同时将该 spills 中间文件的各分区数据长度的数组返回即可
      // Here, we don't need to perform any metrics updates because the bytes written to this
      // output file would have already been counted as shuffle bytes written.
      Files.move(spills[0].file, outputFile);
      return spills[0].partitionLengths;
    } else {
```

```
            final long[ ] partitionLengths;
            //  当存在多个 Spill 中间文件时,根据不同的条件,采用不同的文件合并策略
            if( fastMergeEnabled && fastMergeIsSupported) {
              //由"spark. file. transferTo"配置属性控制,默认为 true。
              if( transferToEnabled) {
                logger. debug( "UsingtransferTo - based fast merge") ;
              //通过 NIO 的方式合并各个 spills 的分区字节数据
//仅在 IO 压缩码和序列化器支持序列化流的合并时安全
                partitionLengths = mergeSpillsWithTransferTo( spills,outputFile) ;
              } else {
                logger. debug( "UsingfileStream - based fast merge") ;
                partitionLengths = mergeSpillsWithFileStream( spills,outputFile,null) ;
              }
            } else {
              //使用 Java FileStreams 文件流的方式进行合并
              partitionLengths = mergeSpillsWithFileStream( spills,outputFile,compressionCodec) ;
            }
            //更新 Shuffle 写数据的度量信息
            writeMetrics. decShuffleBytesWritten( spills[ spills. length - 1]. file. length( ) ) ;
            writeMetrics. incShuffleBytesWritten( outputFile. length( ) ) ;
            returnpartitionLengths;
          }
…
}
}
```

各种合并策略在性能上具有一定的差异，会根据具体的条件采用，主要有基于 NIO 和基于普通文件流合并文件的方式。下面简单描述一下基于文件合并流的处理过程，代码如下。

```
    /** 使用 Java FileStreams 文件流的方式进行合并 */
private long[ ] mergeSpillsWithFileStream(
        SpillInfo[ ] spills,
        FileoutputFile,
        @Nullable CompressionCodec compressionCodec) throws IOException {
      assert( spills. length  > =2) ;
      final intnumPartitions = partitioner. numPartitions( ) ;
      final long[ ]partitionLengths = new long[ numPartitions] ;

      //对应打开的输入流的个数为 spills 的临时文件个数
      finalInputStream[ ] spillInputStreams = new FileInputStream[ spills. length] ;
      OutputStream mergedFileOutputStream = null;

      booleanthrewException = true;
      try {
```




```
        //为每个 spills 中间文件打开文件输入流
        for(int i =0; i < spills.length; i ++ ){
          spillInputStreams[i] = new FileInputStream(spills[i].file);
        }

        //遍历分区
        for(int partition =0; partition < numPartitions; partition ++ ){
          final long initialFileLength = outputFile.length();
          //构建各个分区对应的合并文件输出流
          mergedFileOutputStream =
            new TimeTrackingOutputStream(writeMetrics,new FileOutputStream(outputFile,true));
          if(compressionCodec != null){
              mergedFileOutputStream = compressionCodec.compressedOutputStream(mergedFile-
              OutputStream);
          }

          //依次从各个 Spills 输入流中读取当前分区的数据长度指定个数的字节
          //到各个分区对应的输出文件流中
          for(int i =0; i < spills.length; i ++ ){
            …
          }
          mergedFileOutputStream.flush();
          mergedFileOutputStream.close();
          //将当前写入的数据长度存入返回的数组中
          partitionLengths[partition] = (outputFile.length() - initialFileLength);
        }
        threwException = false;
      } finally {
        …
      returnpartitionLengths;
    }
```

基于 NIO 的文件合并流程基本类似，只是底层采用了 NIO 的技术实现。

Section 7.6 小结

本章内容围绕对集群性能有巨大影响的 Shuffle 实现机制展开，介绍了在 Spark 计算框架中 Shuffle 实现机制的演进过程，以及在各个演进过程中各实现机制的优缺点。之后用 Shuffle 的框架解析作为起点，逐步从整个可插拔框架到各个具体实现机制的内部框架，比较详细地解析各 Shuffle 框架整体组成结构及对外提供的处理流程，并结合各个实现机制内部的关键源代码的解析来加深对 Spark 的 Shuffle 框架的理解。

第 8 章　钨丝计划（Project Tungsten）

Spark 计算框架备受瞩目的原因之一在于它的高性能，因此对性能的追求一直是其主要目标之一，在通过深入分析 Spark 计算框架性能的瓶颈之后，引入了钨丝计划，该计划极大地提高了 Spark 计算框架的性能。

Section 8.1 钨丝计划（Project Tungsten）概述

在 Spark 的近期发展中，最引人关注的是钨丝计划（Project Tungsten）。该计划可以从 Databricks 公司发布的官方博客获取（具体参考地址为 https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html），从该博客中可以知道，目前制约 Spark 计算框架性能的瓶颈主要在于 CPU 与内存，而不是磁盘 I/O 及网络开销——随着带宽增长、SSD 或者磁盘阵列的使用，这部分开销已经不再是 Spark 计算框架的瓶颈。因此，作为 Spark 性能提升的下一阶段的钨丝计划（Project Tungsten）也就因此诞生了。

关于性能方面的详细信息，可以参考上述博客中的相关介绍，以及《Making Sense of Performance in Data Analytics Frameworks》这篇论文。

Project Tungsten 主要包含以下三大方面。

- 内存管理（Memory Management）和二进制处理（Binary Processing）：利用面向应用的语义（application semantics）来更明确地管理内存，同时消除 JVM 对象模型和垃圾回收开销。
- 缓存友好的计算（Cache-aware Computation）：使用算法和数据结构来实现内存分级结构（Memory Hierarchy）。
- 代码生成（Code Generation，CG）：使用代码生成来利用新型编译器和 CPU。

当前钨丝计划的详细信息可以参考官网上的两个开发阶段，如图 8-1 和图 8-2 所示。

Spark / SPARK-7075

Project Tungsten (Spark 1.5 Phase 1)

Agile Board

Details

Type:	Epic	Status:	RESOLVED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	1.5.0
Component/s:	Block Manager, Shuffle, Spark Core, SQL		
Labels:	None		
Epic Name:	Tungsten Phase 1		
Target Version/s:	1.5.0		

图 8-1　Project Tungsten 第一阶段

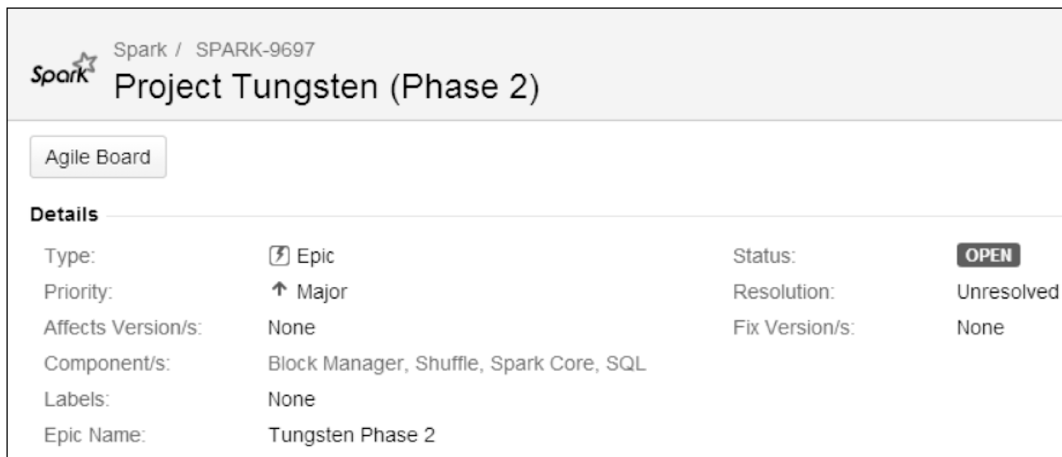


图 8-2　Project Tungsten 第二阶段

更多 Project Tungsten 内容可以跟踪这两个阶段的各个问题（Issue）。

本章的主要目的在于通过解析 Spark 内核部分与 Project Tungsten 有关的内容。对应 Spark SQL 部分与 Project Tungsten 相关的源代码，解析的方式基本是一致的，且部分设计文档在官方资料（尤其是 Issue）中很多都已经给出，而且非常详细，因此本章不再重复这部分内容。

Section 8.2 内存管理模型

8.2.1 现有内存管理的机制

Spark 计算框架是基于 Scala 与 Java 语言开发的，其底层都使用了 JVM。而在 JVM 上运行的应用程序是依赖 JVM 的垃圾回收机制来管理内存的，随着 Spark 应用程序性能的不断提升，JVM 对象和 GC 开销产生的影响（包括内存不足、频繁 GC 或 FullGC）将非常致命。即引进新的 Tungsten 内存管理机制的主要原因在于优化 JVM 在内存方面和 GC 方面的开销。主要包含两个方面。

1. JVM 对象模型（JVM object model）的内存开销

可以通过 Java Object Layout 工具来查看在 JVM 上 Java 对象所占用的内存空间（有兴趣的读者可以参考 http://openjdk.java.net/projects/code-tools/jol/），需要注意的是在 32 位与 64 位的操作系统中，占用空间会有所差异，下面是在 64 位操作系统上对 String 的分析结果。

```
1.  java.lang.String object internals:
2.  OFFSET  SIZE   TYPE DESCRIPTION                VALUE
3.       0    12        (object header)             N/A
4.      12     4 char[] String.value                N/A
5.      16     4    int String.hash                 N/A
6.      20     4    int String.hash32               N/A
7.  Instance size:24 bytes(estimated,the sample instance is not available)
```

```
Space losses:0 bytes internal +0 bytes external =0 bytes total
```

一个简单的 String 对象会额外占用一个 12 字节的 header 和 8 字节的 hash 信息。这是开启（-XX：+UseCompressedOops，默认）指针压缩方式(-XX：+UseCompressedOops)的结果，如果不开启(-XX：+UseCompressedOops)指针压缩(-XX：+UseCompressedOops)，则内存更大，如下所示。

```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION                VALUE
     0    16         (object header)            N/A
    16     8 char[] String.value                N/A
    24     4    int String.hash                 N/A
    28     4    int String.hash32               N/A
Instance size:32 bytes(estimated,the sample instance is not available)
Space losses:0 bytes internal +0 bytes external =0 bytes total
```

其中，Header 会占用 16 个字节的大小，另外 JVM 内存模型会采用 8 字节对齐，因此也可能会再增加一部分内存开销。

以上仅仅是 String 对象中 JVM 内存模型中占用的内存大小，实际计算时需要考虑其内部的引用对象所占的内存，通过分析，char[]中默认的指针压缩情况下会占用 16bytes 内存，因此仅仅是一个空字符串也会占用 2440bytes +1640bytes =4 Kbytes 的内存。

另外在 JVM 内存模型中，为了更加通用，它重新定制了自己的储存机制，使用 UTF-16 方式编码每个字符（2 字节）。可以参考 http://www.javaworld.com/article/2077408/core-java/sizeof-for-java.html,java 对象的内存占用大小，如下所示。

```
// java.lang.Object shell size in bytes:
    public static final int OBJECT_SHELL_SIZE   =8;
    public static final int OBJREF_SIZE         =4;
    public static final int LONG_FIELD_SIZE     =8;
    public static final int INT_FIELD_SIZE      =4;
    public static final int SHORT_FIELD_SIZE    =2;
    public static final int CHAR_FIELD_SIZE     =2;
    public static final int BYTE_FIELD_SIZE     =1;
    public static final int BOOLEAN_FIELD_SIZE  =1;
    public static final int DOUBLE_FIELD_SIZE   =8;
    public static final int FLOAT_FIELD_SIZE    =4;
```

其中 CHAR_FIELD_SIZE 为 2，即每个字符串在 JVM 中采用了 UTF-16 编码，一个字符会占用两个字节。

2. 垃圾回收机制(Garbage collection,GC)的开销

JVM 对象带来的另一个问题是 GC。通常情况下，JVM 内存模型中的 Heap 会分成两大块：Young Generation（年轻代）和 Old Generation（年老代），其中年轻代会有很高的 allocation/deallocation，通过利用年轻代对象的瞬时特性，垃圾收集器可以更有效率地对其

进行管理；年老代的状态则非常稳定。GC 的开销在所有基于 JVM 的应用程序中都是不可忽视的，而且对应的调优也非常繁琐，在类似 Spark 框架这样的基于内存迭代处理的框架中，通常直接在底层对内存进行管理可以极大地提高效率。因此，对应引入 Project Tungsten 也就很合情合理了。

下面的章节会详细解析 Project Tungsten 的内存模型及其源代码实现，并且对基于该模型的 Shuffle 写数据过程中的二进制数据处理也给出了详细解析。

8.2.2 Project Tungsten 内存管理的模型及其源代码的解析

在 2016 年 1 月 4 号发布的 Spark 1.6 中，提出了一个新的内存管理模型，即统一内存管理模型，对应在 Spark 1.5 及之前的版本则使用静态的内存管理模型。关于新的统一内存管理模型，可以参考 https://issues.apache.org/jira/secure/attachment/12765646/unified-memory-management-spark-10000.pdf。在该文档中详细描述了各种可能的设计，以及各设计的优缺点。另外，也可以参考网上对 Spark 内存管理模型解析非常深入的博客 http://0x0fff.com/spark-memory-management/（Alexey Grishchenko），博客内容包含了静态内存模型管理与动态内存模型管理的详细说明。

为了解决现有基于 JVM 托管方式的内存模型所存在的缺陷，Project Tungsten 设计了一套新的内存管理机制。在新的内存管理机制中，Spark 的 Operation 可以直接使用分配的 Binary Data（二进制数据）而不是 JVM Objects。避免了数据处理过程中不必要的序列化与反序列化的开销，同时基于 Off-Heap 方式管理内存，降低了 GC 所带来的开销。

Project Tungsten 通过 sun.misc.Unsafe 来管理内存，关于 sun.misc.Unsafe（从命名上可知该工具不能滥用）及其使用等内容，可以参考官网文档 http://www.docjar.com/docs/api/sun/misc/Unsafe.html。在此主要分析 Project Tungsten 中的内存管理模型的具体实现。

1. Project Tungsten 内存模型

Project Tungsten 内存管理模型主要的类图结构如图 8-3 所示。

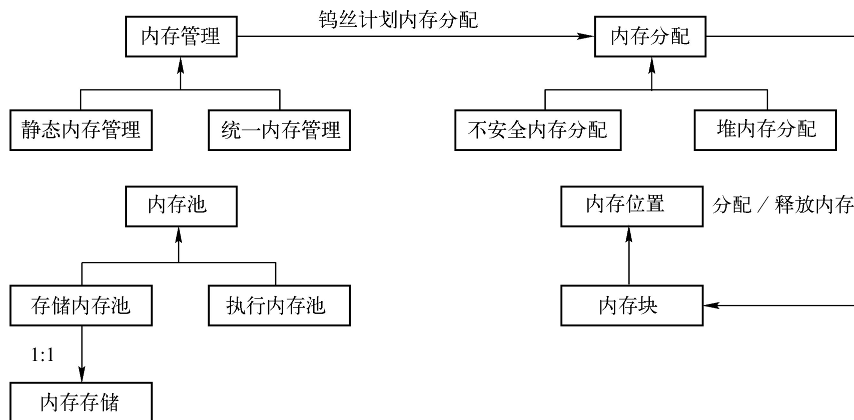


图 8-3　Project Tungsten 内存管理模型主要的类图结构

在图 8-3 中，基类 MemoryManager 封装了静态内存管理模型与统一内存管理模型，即分

别对应两个具体实现子类 StaticMemoryManger 与 UnitedMemoryManager。对应的内存分配由 MemoryManager 的成员 tungstenMemoryMode 决定，即由基类 MemoryAllocator 负责具体内存分配，对应 Off-Heap 与 On-Heap 两种内存模式，分别实现了两个具体子类 UnsafeMemoryAllocator 与 HeapMemoryAllocator。MemoryAllocator 提供了 Allocate 和 Free 两个成员函数来提供内存的分配与释放，分配的内存以 MemoryBlock 来表示。

另外，根据内存使用目的的不同，将内存分为两大部分：Storage 和 Execution，对应的以 MemoryPool 的两个具体实现子类 StorageMemoryPool 与 ExecutionMemoryPool 对其进行管理。实际上除了这两部分，总的内存还包括为系统预留的 OtherMemory。

关于内存分类及其对应管理的主要类之间的关系，可以通过图 8-4 来描述。

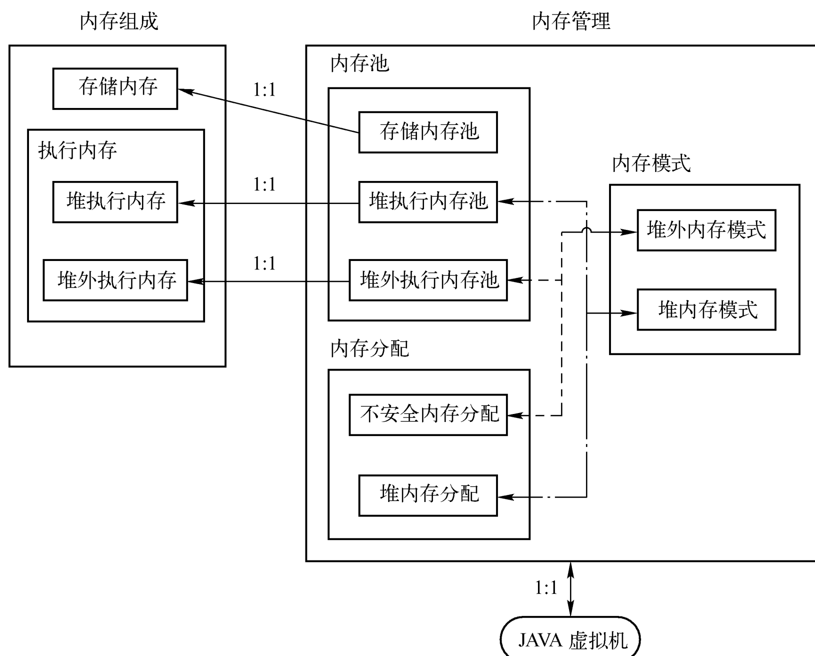


图 8-4　内存分类及其对应管理的主要类之间的关系

在 Worker 上运行的每个 Execution 进程（抽象描述，实际对应各部署场景下的具体 ExecutorBackend 实现子类），对应由一个 MemoryManager 负责管理其内存，即图 8-4 中 MemoryManager 与 JVM 的对应关系为 1∶1。

Storage 部分的内存由 StorageMemoryPool 负责管理，Execution 部分的内存根据不同的内存模式(MemoryMode)分为 on-heap 与 off-heap 两种，分别由 onHeapExecutionMemoryPool 与 offHeapExecutionMemoryPool 进行管理。管理内存主要是通过内存使用量进行控制，不涉及内存的分配与释放。

2. MemoryManager 的实现及其源代码解析

MemoryManager 目前实现了两种具体的内存管理模型，从 Spark 1.6 版本开始，默认使用统一内存管理模型，对应的配置属性为"spark.memory.useLegacyMode"，控制代码位于 SparkEnv 类中，代码如下。

```
// Spark 1.5 及之前的版本所使用的内存管理模型对应配置属性
//"spark.memory.useLegacyMode"。当前默认为 false
  val useLegacyMemoryManager = conf.getBoolean("spark.memory.useLegacyMode",false)
      val memoryManager:MemoryManager =
        if(useLegacyMemoryManager){
          //使用静态内存管理模型
          new StaticMemoryManager(conf,numUsableCores)
        } else {
//使用统一内存管理模型
          UnifiedMemoryManager(conf,numUsableCores)
        }
```

以上是选择具体采用哪种内存管理模型的代码，下面开始分析与内存管理相关的源代码，首先查看 MemoryManager 的注释，代码如下。

```
/**
  * 内存管理的抽象接口,用于指定如何在 Execution 与 Storage 间共享内存
  * Execution Memory 是指用计算的内容,包括 Shuffles、joins、sorts 及 aggregations
  * Storage Memory 是指用于缓存或内部数据传输过程中所使用的内存
  * MemoryManager 与 JVM 进程的对应关系为 1:1。即一个 JVM 进程中的内存由一个
  * MemoryManager 进行管理
  */
private[spark] abstract class MemoryManager(
…
```

在 MemoryManager 类中提供的内存分配与释放的几个主要接口如下。

- Storage 部分内存的分配与释放接口：acquireStorageMemory、acquireUnrollMemory、releaseStorageMemory 和 releaseUnrollMemory。
- Execution 部分内存的分配与释放接口：acquireExecutionMemory 和 releaseExecutionMemory。

内存具体分配与释放的实现由 MemoryManager 的具体子类提供。

两大实现子类（StaticMemoryManager 和 UnifiedMemoryManager）的主要差别在于 Storage 与 Execution 内存之间的边界是静态的还是动态可变的，下面分别简单描述两大子类的实现细节。

StaticMemoryManager 类的注释如下所示。

```
/**
  * A [[MemoryManager]] that statically partitions the heap space into disjoint regions.
  *
  * The sizes of the execution and storage regions are determined through
  *'spark.shuffle.memoryFraction' and' spark.storage.memoryFraction' respectively. The two
  * regions are cleanly separated such that neither usage can borrow memory from the other.
```

```
 * 静态划分 Storage 与 Execution 内存之间的边界的一种内存管理实现
  * Storage 与 Execution 内存大小分别由配置属性 spark. shuffle. memoryFraction 与
 * spark. storage. memoryFraction 各自指定,由于是静态划分边界,因此这两者之间不能
  * 互相借用多余的内存
  */
private[spark] class StaticMemoryManager(
```

静态内存管理模型中各部分内存的分配可以通过以下几个接口或成员变量查看。

1）maxUnrollMemory：unroll 过程中可用的内存，占最大可用 Storage 内存的 0.2（占比）。

2）getMaxStorageMemory：获取分配给 Storage 使用的最大内存大小。

3）getMaxExecutionMemory：获取分配给 Execution 使用的最大内存大小。

其中，getMaxStorageMemory 对应用于 Storage 的最大内存，具体配置如下。

```
  private def getMaxStorageMemory(conf:SparkConf):Long = {
   val systemMaxMemory = conf. getLong("spark. testing. memory",Runtime. getRuntime. maxMemory)
   val memoryFraction = conf. getDouble("spark. storage. memoryFraction",0. 6)
   val safetyFraction = conf. getDouble("spark. storage. safetyFraction",0. 9)
   (systemMaxMemory  * memoryFraction  * safetyFraction). toLong
 }
```

其中配置属性 spark. storage. memoryFraction 表示 Storage 内存占用全部内存（除预留给系统的内存外）的占比，spark. storage. safetyFraction 对应为 Storage 内存的安全系数。

相应的，getMaxExecutionMemory 方法指明了用于 Execution 内存的相关配置属性，与 Storage 内存一样包含占总内存的占比（0.2）及对应的安全系数。

另外，除了 Storage 内存与 Execution 内存占用的 0.6 + 0.2 之外的剩余内存，作为系统预留内存。

通过 StaticMemoryManager 类简单分析静态内存管理模型后，继续查看统一内存管理模型，首先查看其类注释。

```
/**
  * UnifiedMemoryManager:是 MemoryManager 的一个具体子类,实现 Storage 与
  * Execution 间软边界(即动态边界)的内存管理模式
  * 动态边界意味着 Storage 与 Execution 的内存是可以相互借用的
  * Storage 与 Execution:共享的内存通过配置 spark. memory. fraction 进行设置
  * 两者间的内存分配通过配置 spark. memory. storageFraction 进行设置
  * 内存借用:
  *   1) Storage 可以借用 Execution 中空闲的内存,但在 Execution 执行需要内存时
 *    会被回收(回收缓存内存,直到满足执行申请的内存)
  *   2)同样,Execution 也可以借用 Storage 中空闲的内存。但是,出于实现的复
  *   杂性考虑(具体可以参考前面给出的官方设计文档)
 * unified - memory - management - spark - 10000. pdf),
```

```
 *        借用的内存永远不会因为 Storage 的需要而进行回收
 */
private[spark] class UnifiedMemoryManager private[memory](
…
```

UnifiedMemoryManager 与 StaticMemoryManager 一样实现了 MemoryManager 的几个内存分配和释放的接口，对应分配与释放接口的实现，在 StaticMemoryManager 中相对比较简单，而在 UnifiedMemoryManager 中，由于考虑到动态借用的情况，实现相对比较复杂，具体细节可以参考官方提供的统一内存管理设计文档及相关源代码，比如针对各个 Task 如何保证其最小分配的内存（最少为 1/2N，其中 N 表示当前活动状态的 Task 个数，最大的 Task 个数可以从 Executor 分配的内核个数/每个 Task 占用的内核个数得到）等。

下面简单分析一下统一内存管理模型中，Storage 内存与 Execution 内存等相关的配置。

查看方法，具体代码如下。

```
  /**

 *返回 Execution 与 Storage 共享的最大内存
   */
  private def getMaxMemory(conf:SparkConf):Long = {
    val systemMemory = conf.getLong("spark.testing.memory",Runtime.getRuntime.maxMemory)
//系统预留的内存大小,默认为 300MB
//结合下面最小系统内存的限定条件(硬编码方式,因此修改需要重新编译),
    //可以人为地修改该配置进行测试
    val reservedMemory = conf.getLong("spark.testing.reservedMemory",
      if(conf.contains("spark.testing"))0 else RESERVED_SYSTEM_MEMORY_BYTES)
//当前最小的内存需要 300×1.5,即 450 MB,不满足该条件时会报错退出
val minSystemMemory = reservedMemory  * 1.5
    if(systemMemory < minSystemMemory) {
      throw new IllegalArgumentException(s"System memory$systemMemory must " +
        s"be at least$minSystemMemory. Please use a larger heap size.")
    }
    val usableMemory = systemMemory - reservedMemory
    //当前 Execution 与 Storage 共享的最大内存占比默认为 0.75,即
    // Execution 与 Storage 内存为可用内存的 0.75;
    //用户内存为可用内存的(1 -0.75) =0.25
    //由于前面提到的 Alexey Grishchenko 给出的博客上对统一内存管理模型给出了
//非常详细、深入的解析,建议直接参考博客内容,同时结合此处源代码进行理解
    val memoryFraction = conf.getDouble("spark.memory.fraction",0.75)
    (usableMemory  * memoryFraction).toLong
  }
```

另外，虽然 Execution 与 Storage 之间共享内存，但仍然存在一个初始边界值，参考伴生对象 UnifiedMemoryManager 的 apply 工厂方法，具体代码如下。



```
def apply(conf:SparkConf,numCores:Int):UnifiedMemoryManager = {
  val maxMemory = getMaxMemory(conf)
  new UnifiedMemoryManager(
    conf,
    maxMemory = maxMemory,
// 通过配置属性"spark.memory.storageFraction",可以设置 Execution 与 Storage
    //共享内存的初始边界值,即默认初始化时,各占总内存的一半
    storageRegionSize =
      (maxMemory * conf.getDouble("spark.memory.storageFraction",0.5)).toLong,
    numCores = numCores)
  }
```

另外需要注意的是，前面 Execution 内存指的是 ON_HEAP 部分的内存，在 ProjectTungsten 中引入了 OFF_HEAP（堆外）内存，这部分内存大小的设置在基类 MemoryManager 中，对应代码如下。

```
//根据传入的内存大小或配置属性分别设置内存池管理的内存初始大小
//1. Storage 部分的内存池初始大小设置
storageMemoryPool.incrementPoolSize(storageMemory)
// 2. ON_HEAP 部分的 Execution 内存池初始大小设置
onHeapExecutionMemoryPool.incrementPoolSize(onHeapExecutionMemory)
// 3. 从配置属性中读取 OFF_HEAP 内存池的初始内存大小
offHeapExecutionMemoryPool.incrementPoolSize(conf.getSizeAsBytes("spark.memory.offHeap.size",0))
```

当需要使用 OFF_HEAP 内存时，需要注意的是，除了需要修改 OFF_HEAP 内存池（offHeapExecutionMemoryPool）的内存初始值（默认为 0）外，还需要打开对应的控制开关，具体代码参考内存分配 MemoryManager 中内存模式的设置（该内存模式可以控制用于内存分配 MemoryAllocator 的具体子类），对应代码如下。

```
final val tungstenMemoryMode:MemoryMode = {
//当需要使用 OFF_HEAP 内存模式时,需要通过"spark.memory.offHeap.enabled"
    //配置属性打开开关,然后通过"spark.memory.offHeap.size"配置属性
    //指定 OFF_HEAP 的内存大小
    if(conf.getBoolean("spark.memory.offHeap.enabled",false)){
      require(conf.getSizeAsBytes("spark.memory.offHeap.size",0) >0,
        "spark.memory.offHeap.size must be >0 when spark.memory.offHeap.enabled == true")
        MemoryMode.OFF_HEAP
      } else {
        MemoryMode.ON_HEAP
      }
  }
```

从图 8-4 中可以看出，Execution 内存根据不同的内存模式（ON_HEAP 或 OFF_HEAP）可以有两种内存池管理方式，可以查看一下 Execution 内存分配的方法，关键代码如下。

```
override private[memory] def acquireExecutionMemory(
    numBytes:Long,
    taskAttemptId:Long,
    memoryMode:MemoryMode):Long = synchronized {
  assert(onHeapExecutionMemoryPool.poolSize + storageMemoryPool.poolSize == maxMemory)
  assert(numBytes >= 0)
  memoryMode match {
    case MemoryMode.ON_HEAP =>
…
       //当内存模式为 ON_HEAP 时,使用 onHeapExecutionMemoryPool 内存池来管理
       onHeapExecutionMemoryPool.acquireMemory(
       numBytes,taskAttemptId,maybeGrowExecutionPool,computeMaxExecutionPoolSize)

     case MemoryMode.OFF_HEAP =>
       //当内存模式为 OFF_HEAP 时,使用 offHeapExecutionMemoryPool 内存池来管理
       offHeapExecutionMemoryPool.acquireMemory(numBytes,taskAttemptId)
    }
  }
```

MemoryMode 是二选一，因此在启动 OFF_HEAP 内存模式时，可以将 Storage 的内存占比（对应配置属性"spark.memory.storageFraction"）设置得高一点，虽然在具体分配过程中 Storage 也可以向 ON_HEAP 这部分 Execution 借用内存。

关于内存池部分，下面给出主要类的类图，具体实现可以结合类图阅读源代码来加深理解，主要类图如图 8-5 所示。

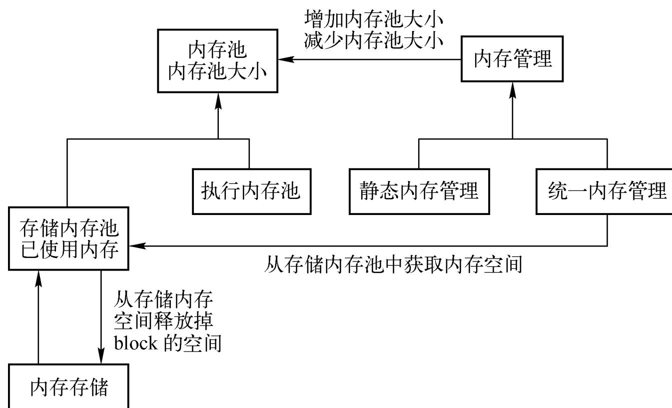


图 8-5　内存池相关类图

主要通过内部池大小和使用的内存大小等进行控制，对应统一内存管理模型，需要考虑借用等具体实现（关键代码可以查看 UnitedMemoryManager 对 StorageMemoryPool 类的 shrinkPoolToFreeSpace 方法的调用）。

以上是对 Tungsten 的两种内存管理模型的简单解析。下面开始对内存管理模型的内部组织结构进行解析。

3. Project Tungsten 内存管理模型中对内存描述的封装

关于 Project Tungsten 的相关内容，可以参考 https://github.com/hustnn/TungstenSecret。其中对 Page Table 给出了描述非常详细的说明图。

下面从最基本的源代码开始逐步分析内存管理模型中内存描述的封装，主要包含内存地址的封装和内存块的封装，分别对应 MemoryLocation 和 MemoryBlock。

在 Project Tungsten 中，为了统一管理 ON_HEAP 和 OFF_HEAP 两种内存模式，引入了统一的地址表示形式，即通过 MemoryLocation 类来表示 ON_HEAP 或 OFF_HEAP 两种内存模式下的地址。

首先查看该类的注释信息，具体如下所示。

```
/**
 * A memory location. Tracked either by a memory address(with OFF_HEAP allocation),
 * or by an offset from a JVM object(in - heap allocation).
* 一个内存地址。用于跟踪 OFF_HEAP 模式下的内存地址或 ON_HEAP 模式下的内存地址
 */
public class MemoryLocation {
```

当使用 OFF_HEAP 内存模式时，内存地址可以通过 64 位的绝对地址来描述，相应的，当使用 ON_HEAP 内存模式时，由于 GC 过程中会对堆（heap）内存进行重组，因此地址的定位需要通过对象在堆内存的引用及在该对象内的偏移量来表示，此时便需要对象引用和一个偏移量来表示内存地址。

因此，在 MemoryLocation 中定义了两个成员变量，具体代码如下。

```
@ Nullable
Object obj;
long offset;
```

对应两种不同的内存模式，两个成员变量的描述如下。

1）OFF_HEAP 内存模式：obj 为 null，地址由 64 位的 offset 唯一标识。

2）ON_HEAP 内存模式：obj 为堆中该对象的引用，offset 对应数据在该对象中的偏移量。

由以上分析可知，通过 MemoryLocation 类可以统一定位一个 OFF_HEAP 和 ON_HEAP 两种内存模式下的内存地址。

对应 MemoryLocation 类的继承子类为 MemoryBlock，顾名思义，该子类表示一个内存块，无论是 OFF_HEAP 还是 ON_HEAP 内存模式，在 Project Tungsten 内存管理时，都使用一块连续的内存空间来存储数据，因此即使是在 ON_HEAP 模式下，也可以降低 GC 的开销。下面来看一下 MemoryBlock 类的注释信息，具体如下。

```
/**
 * A consecutive block of memory,starting at a {@linkMemoryLocation} with a fixed size.
 * 一个连续的内存块,继承自描述内存地址的 MemoryLocation 类,同时提供内存块的
 * 大小
 */
public classMemoryBlock extends MemoryLocation {
```

📖 补充：在代码复用方式上存在两种形式：继承与组合。目前，在 MemoryBlock 中使用继承的方式包含内存块的地址信息。在实现上，也可以采用组合这种复用方式，指定内存块的地址，以及内存块本身的内存大小。

下面简单介绍一下 MemoryBlock 类中除了继承自 MemoryLocation 类之外的部分成员。

1） private final long length：表示内存块的长度。

2） public int pageNumber：表示内存块对应的 page 号。

3） public static MemoryBlock fromLongArray （final long [ ] array）：这是提供的一个将 long 型数组转换为 MemoryBlock 内存块的接口。

在提供了内存块之后，下一步就是如何去组织这些内存块，在 Project Tungsten 中采用了类似操作系统的内存管理模式，即使用 Page Table 方式来管理内存。因此，下面将对 Page Table 管理方式进行解析。

4. Project Tungsten 内存管理模型中的内存组织和管理模式

Spark 是一个技术框架，数据以分区粒度进行处理，即每个分区对应一个处理的任务（Task），因此内存的组织与管理等可以通过与 Task 一一对应的 TaskMemoryManager 来理解。

下面首先给出 TaskMemoryManager 与 MemoryManager 间的关系图，如图 8-6 所示。

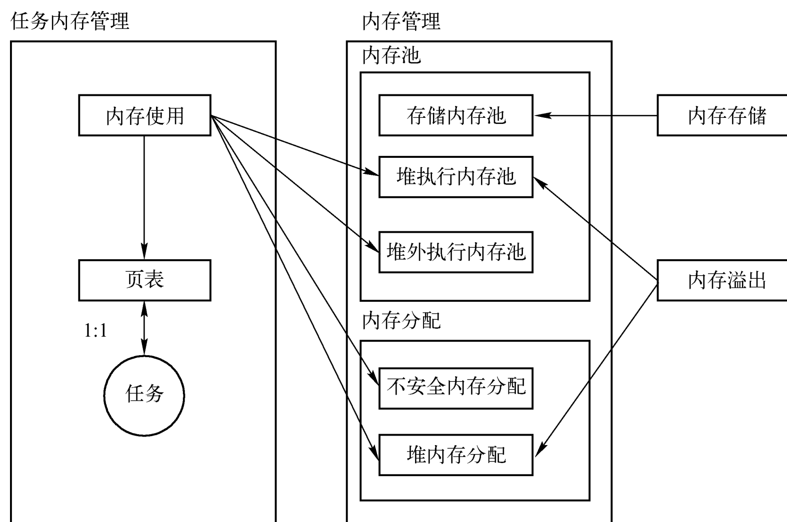


图 8-6　TaskMemoryManager 与 MemoryManager 的关系图

在图 8-6 中，各个 MemoryConsumer 是具体处理时需要使用（消耗）内存块的实体，MemoryConsumer 通过 TaskMemoryManager 提供的接口向 MemoryManager 申请或释放内存资源，即申请或释放内存块。TaskMemoryManager 类中会管理全部 MemoryConsumer，并对这些内存消耗实体所申请的内存块进行组织与管理，具体是通过 PageTable 的方式来实现。

首先查看类的注释信息，原注释信息比较多，在此仅给出简单的中文描述，具体代码如下。

```
/**
 * Manages the memory allocated by an individual task.
 * …
 * 管理为单个 Task 所分配的内存
 * 内存地址在不同的内存模式下的表示：
 *   1. OFF_HEAP:直接使用 64 位表示内存地址
 *   2. ON_HEAP:通过 base object 和该对象中 64 位的偏移量来表示
 * 通过封装类 MemoryBlock 统一表示内存块信息：
 *   1. OFF_HEAP:MemoryBlock 的 base object 为 null,偏移量对应 64 位的绝对地址
 *   2. ON_HEAP:MemoryBlock 的 base object 保存对象的引用(该引用可以由 page 的索引从 pageTable 获取)
 * 偏移量对应数据在该对象中的偏移量
 *
 * 通过这两种内存模式对应的编码方式,最终对外提供的编码格式为 13bit - pageNumber + 51bit - offset
 */
public classTaskMemoryManager {
```

下面从 3 个方面对 TaskMemoryManager 进行分析：包含内存地址的编码与解码、PageTable 的组织与管理，以及内存的分配与释放。

（1）内存地址的编码与解码

从 TaskMemoryManager 类的注释部分可以知道，OFF_HEAP 与 ON_HEAP 两种内存模式最终对外都是采用一致的编码格式，即对应 13 位的 pageNumber（页码）和 51 位的 offset（偏移量），可以通过图 8-7 来描述对应的编码方式。

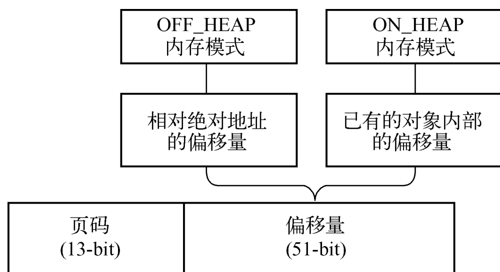


图 8-7　Page 的编码方式

下面分别对 TaskMemoryManager 类中与编码和解码相关的几个接口进行解析，编码接口主要有两个，encodePageNumberAndOffset 和 decodePageNumber 其源代码与解析如下所示。

```
/**
 * Given a memory page and offset within that page,encode this address into a 64-bit long.
 * This address will remain valid as long as the corresponding page has not been freed.
 *
 * @param page a data page allocated by {@link TaskMemoryManager#allocatePage}/
 * @param offsetInPage an offset in this page which incorporates the base offset. In other
 *   words,   this should be the value that you would pass as the base offset into an
 *                    UNSAFE call(e.g. page.baseOffset() + something).
 * @return an encoded page address.
  *
  * 将针对某个 Page 的地址进行编码：
  * ON_HEAP:offsetInPage 是针对 base object 的偏移量
  * OFF_HEAP:此时,offsetInPage 是绝对地址,因此编码到 Page 方式的地址时,
  * 需要将绝对地址转换为相对于已有的 Page(MemoryBlock)中的绝对
  * 地址 offset 的相对地址。最后将得到的两个偏移量和 Page Number 一起组装到 13+51 bits
 的 64 位中。
  */
 public long encodePageNumberAndOffset(MemoryBlock page,long offsetInPage){
   if(tungstenMemoryMode == MemoryMode.OFF_HEAP){
     //如果是 OFF_HEAP,则对应的 offsetInPage 为 64 位的绝对地址,需要转换为 Page
     //编码能容纳的 51 位编码中,因此此时需要将其转换为 Page 内的相对地址,
     //即页内的偏移地址



     offsetInPage -= page.getBaseOffset();
   }
   return encodePageNumberAndOffset(page.pageNumber,offsetInPage);
 }

 @VisibleForTesting
 public static long encodePageNumberAndOffset(intpageNumber,long offsetInPage){
   assert(pageNumber != -1):"encodePageNumberAndOffset called with invalid page";
 //将 13 位的页码与 51 位的页内偏移量组装成 64 位的编码地址
   return(((long)pageNumber) << OFFSET_BITS) | (offsetInPage & MASK_LONG_LOWER_
   51_BITS);
 }
```

通过 pageNumber 可以找到最终的 Page，Page 内部会根据 OFF_HEAP 或 ON_HEAP 两种模式分别存储 Page 对应内存块的起始地址（或对象内偏移地址），因此编码后的地址可以通

过查找到 Page，最终解码出原始地址。解码的源代码及其解析如下所示。

```
@ VisibleForTesting
public static intdecodePageNumber( long pagePlusOffsetAddress) {
 //通过 13 位掩码解析出编码地址中的页码信息,即对应的高 13 位内容
   return( int)( ( pagePlusOffsetAddress & MASK_LONG_UPPER_13_BITS) >>> OFFSET_BITS);
}

private static longdecodeOffset( long pagePlusOffsetAddress) {
//通过 51 位掩码解析出编码地址中的页码信息,即对应的低 51 位内容
   return( pagePlusOffsetAddress & MASK_LONG_LOWER_51_BITS);
 }
```

在 TaskMemoryManager 类中还另外提供了针对 ON_HEAP 内存模式下获取 base object 的接口，对应的源代码及其解析如下所示。

```
/ **
 * Get the page associated with an address encoded by
* 获取编码地址相关的 base object
 * {@ linkTaskMemoryManager#encodePageNumberAndOffset( MemoryBlock,long) }
 * /
public ObjectgetPage( long pagePlusOffsetAddress) {
   if( tungstenMemoryMode == MemoryMode. ON_HEAP) {
     //首先从地址中解析出页码
     final intpageNumber = decodePageNumber( pagePlusOffsetAddress);
     assert( pageNumber >=0 && pageNumber < PAGE_TABLE_SIZE);
     //根据页码从 pageTable 变量中获取对应的内存块
     finalMemoryBlock page = pageTable[ pageNumber];
     assert( page ! = null);
     assert( page. getBaseObject( )! = null);
//获取内存块对应的 BaseObject
     return page. getBaseObject( );
   } else {
     // OFF_HEAP 内存模式下,MemoryBlock 只需要保存一个绝对地址,因此对应的
     // base object 为 null
     return null;
   }
 }
```

（2）PageTable 的组织与管理

在分析这部分内容之前，先看一下 Page Table 方式进行组织与管理描述图，如图 8-8 所示。

在图 8-8 中，右侧是分配的内存块，即当前需要管理的 Page。在 TaskMemoryManager 中，通过 Page Table 来存放内存块，同时，通过在变量 allocatedPages 中指定值为 Page Number（页码）的下标（索引）对应的值是否为 1，来表示当前 Page Number 对应的 Page Table

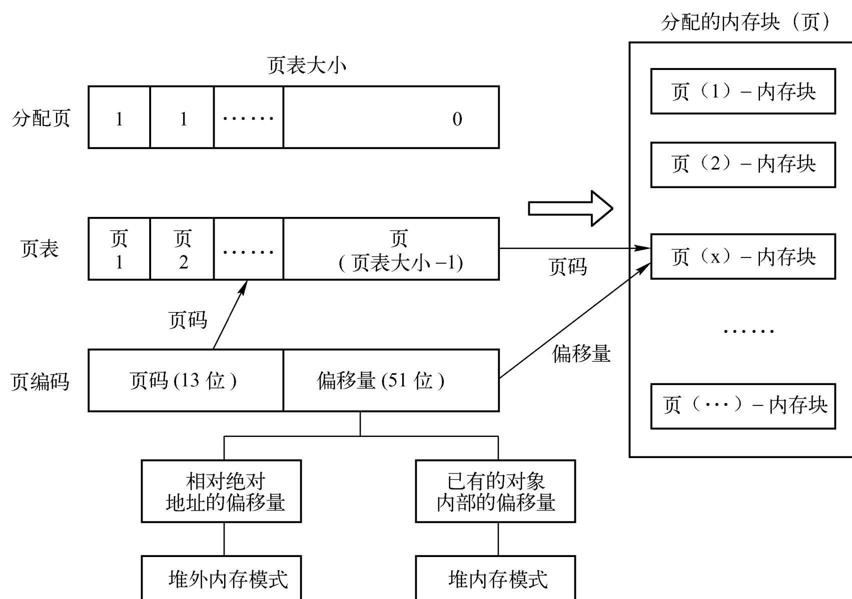


图 8-8　PageTable 组织与管理描述图

中的 Page 是否已经存放了对应的内存块，即每当分配到一个内存块时，从 allocatedPages 获取一个值为 0 的位置（页码），并将该位置作为内存块放入到 Page Table 中的位置。

简单来说，就是通过 allocatedPages 中各个位置上的值为 1 或 0 来判断在 Page Table 中相同位置是否已经放置了内存块（Page）。

而对应在 Page Table 中已经存放的内存块，实际上就是对应了右侧已经分配的内存块。

当针对一个 Page Encode（页地址编码）时，首先从中获取 Page Number，根据该值从 Page Table 中获取确定的内存块（MemoryBlock 或 Page），找到确定内存块之后，再通过页地址编码中的 offset（具体两种内存模式下的概念如图 8-8 所示）确定内存块中的相关偏移量。如果是 OFF_HEAP，则该 offset 是相对于内存块（从前面分析可知，内存块本身的信息也与内存模式相关）中的绝对地址的相对地址；如果是 ON_HEAP，则该 offset 是相对于内存块的 base object 中的偏移量。

相关的源代码主要涉及 TaskMemoryManager 类的两个成员变量，如下所示。

```
//对应图 8-8 中的 Page Table
private final MemoryBlock[] pageTable = new MemoryBlock[PAGE_TABLE_SIZE];

//对应图 8-8 中的 allocatedPages
private final BitSet allocatedPages = new BitSet(PAGE_TABLE_SIZE);
```

PageTable 的组织与管理中关于页码的偏移量已经在上一部分给出了详细描述，而对应的具体管理操作则与实际的内存分配与解析部分相关。通过图 8-8 对大致的管理有一定的概念后，再继续通过内存分配与解析部分来详细解析具体的管理细节。

（3）内存分配与解析

关于这部分内容，主要参考 allocatePage 与 freePage 两个方法，对应 allocatePage 内部如

何申请内存，以及申请内存时采用的 spill 策略等细节，大家可以继续深入，比如通过查看 acquireExecutionMemory 的具体源代码来加深理解。

allocatePage 方法的源代码及其解析如下所示。



```
/ **
 * Allocate a block of memory that will be tracked in theMemoryManager' s page table;this
 * is intended for allocating large blocks of Tungsten memory that will be shared between
 * operators.
 * Returns `null` if there was not enough memory to allocate the page.  May return a page
 * that contains fewer bytes than requested,so callers should verify the size of returned
 * pages.
 * 分配一块内存,并通过 MemoryManager(实际上是在 TaskMemoryManager 中)的
 * Page Table 进行跟踪;分配的是 Execution 部分的内存
 * Project Tungsten 的内存包含 OFF_HEAP 和 ON_HEAP 两种模式,由底层
 * tungstenMemoryMode(在 MemoryManager 中设置)控制具体分配的
* MemoryAllocator 子类
 */
public MemoryBlock allocatePage(long size,MemoryConsumer consumer){
  //页大小的限制
  if(size > MAXIMUM_PAGE_SIZE_BYTES){
    throw new IllegalArgumentException(
      "Cannot allocate a page with more than" + MAXIMUM_PAGE_SIZE_BYTES + "bytes");
  }

  //特定的内存消费者 consumer 以指定的内存模式 tungstenMemoryMode
  //申请一定的内存量
  long acquired = acquireExecutionMemory(size,tungstenMemoryMode,consumer);
  if(acquired  <=0){
    return null;
  }

  final intpageNumber;
  synchronized(this){
    //获取当前未被占用的页码
    pageNumber = allocatedPages. nextClearBit(0);
    if(pageNumber >= PAGE_TABLE_SIZE){
      releaseExecutionMemory(acquired,tungstenMemoryMode,consumer);
      throw new IllegalStateException(
        "Have already allocated a maximum of " + PAGE_TABLE_SIZE + " pages");
    }
    //设置该页码已经被占用(即设置对应页码位置的值)
    allocatedPages. set(pageNumber);
```

```
        }

        //开始通过 MemoryAllocator 真正分配内存
        //注意:acquireExecutionMemory 中通过 ExecutionMemoryPool 进行分配时,
        //仅仅是内存使用大小上的控制,并没有真正分配内存
        //有兴趣的话,可以查看对 acquireExecutionMemory 的调用点(其中
        //可以指定与 tungstenMemoryMode 不同的其他内存模式,
        //此时是不存在真正的内存分配的)
        finalMemoryBlock page = memoryManager.tungstenMemoryAllocator().allocate(acquired);

        //分配得到内存块之后,会设置该内存块对应的 pageNumber,
        //即此时设置 MemoryBlock 在其管理的 Page Table 中的位置
        page.pageNumber = pageNumber;
        pageTable[pageNumber] = page;
        if(logger.isTraceEnabled()){
          logger.trace("Allocate page number {} ({} bytes)",pageNumber,acquired);
        }
        return page;
    }
```

其中，MAXIMUM_PAGE_SIZE_BYTES 是页内数据量大小的限制，从之前 MemoryBlock 提供的从 long 型数组转换得到 MemoryBlock 接口，可以知道当前连续的内存块是通过 long 型数组来获取的，因此对应的内存块的大小也会受到数组的最大长度的限制。

至于对应在具体的处理过程中，对页内的数据量大小是否还有其他限制，可以参考具体的处理细节，下一节会给出一个具体处理过程的源代码解析，其中会包含这部分内容。

由于内存分配的细节比较多，这里给出主要的过程描述。

1）首先通过 acquireExecutionMemory 方法向 ExecutionMemoryPool 申请内存（根据统一或静态两种具体实现给出）：这一部分主要是判断当前可用的内存是否满足申请需求，并根据申请结果修改当前内存池可用的内存信息（实际是当前使用内存量信息）。

2）从当前 Page Table 中找出一个可用位置，用于存放所申请的内存块（MemoryBlock 或 Page）。

3）准备好前两步后，开始通过 MemoryAllocator 真正分区内存块。

4）将分配的内存块放入 Page Table。

在整个过程中，allocatedPages 与 pageTable 这两个成员变量的使用是体现 Page Table 组织与管理的关键所在。

下面解析 freePage 的源代码，如下所示。

```
    /**
     * Free a block of memory allocated via {@linkTaskMemoryManager#allocatePage}.
     * 更新 Page Table 相关信息,通过 MemoryAllocator 释放 Page 的内存,
```

```
     * 最后通过 MemoryManager 修改 ExecutorManagerPool 中的内存使用量(即释放)
     */
    public void freePage(MemoryBlock page,MemoryConsumer consumer){
      //首先确认当前释放的内存块位 Page Table 的管理中,即页码必须有效
      assert(page.pageNumber != -1):
        "CalledfreePage()on memory that wasn't allocated with allocatePage()";
      assert(allocatedPages.get(page.pageNumber));
      pageTable[page.pageNumber] = null;

      //allocatedPages 是控制 pageTable 中对应位置是否可用的,
      //需要考虑释放与分配时的并发性,因此需要同步处理
      synchronized(this){
      allocatedPages.clear(page.pageNumber);
      }
      if(logger.isTraceEnabled()){
        logger.trace("Freed page number {}({} bytes)",page.pageNumber,page.size());
      }

      //通过当前内存模式对应的 MemoryAllocator 真正释放该内存块
      long pageSize = page.size();
  memoryManager.tungstenMemoryAllocator().free(page);

      //对应 ExecutionMemoryPool 部分的内存释放,
      //参考前面 acquireExecutionMemory 解析一起了解
      releaseExecutionMemory(pageSize,tungstenMemoryMode,consumer);
    }
```

释放 Page 的逻辑实际上可以参考申请 Page，大部分都是步骤相反而已。

Section 8.3 基于内存管理模型的 Shuffle 二进制数据处理

就目前来说，Project Tungsten 的二进制数据处理主要是用在 Shuffle 和 SQL 的 aggregation（聚合）（以及其他一些操作）数据上，像为其他非 JVM 的本地类库（如 C++类库）等提供内存访问等操作，仍然还在实现中（有兴趣的读者可以参考 https://issues.apache.org/jira/browse/SPARK-10399 部分）。

本书基于 Shuffle 过程，解析在源代码中具体如何使用 Project Tungsten 来处理数据，对应其他操作的处理细节，可以参考对应 issues 及其设计文档（部分提供了非常详细的设计文档）。比如在聚合方面使用 Project Tungsten 内存模型的详细设计可以参考 https://issues.apache.org/jira/browse/SPARK-7080，对应的设计文档为 https://github.com/apache/spark/pull/5725。读者可以基于这些设计文档，然后参考本节的源代码解析过程来加深理解。

在7.5 Tungsten Sorted Based Shuffle一节写数据的源代码解析中已经提到，在写数据时会使用一个外部排序器ShuffleExternalSorter对Shuffle数据进行排序，该外部排序器中的数据处理就是建立在Project Tungsten内存模型基础之上的。因此本节继续深入解析Shuffle写数据的过程，从Project Tungsten内存模型的使用角度来结合源代码进行解析。

首先，从前面对TaskMemoryManager的源代码解析可以知道，所有内存申请与释放的请求都是通过MemoryConsumer来提交的，因此首先需要了解在Shuffle的写过程中，外部排序器ShuffleExternalSorter与内存消费者MemoryConsumer之间的关系。为了了解这一点，可以先查看ShuffleExternalSorter类的注释与类定义，具体源代码如下。

```
/**
 * An external sorter that is specialized for sort-based shuffle.
 * <p>
 * Incoming records are appended to data pages. When all records have been inserted(or
 * when the current thread's shuffle memory limit is reached),the in-memory records are
 * sorted according to their partition ids(using a {@link ShuffleInMemorySorter}). The
 * sorted records are then written to a single output file(or multiple files,if we've spilled). The
 * format of the output files is the same as the format of the final output file written by
 * {@link org.apache.spark.shuffle.sort.SortShuffleWriter};each output partition's records are
 * written as a single serialized,compressed stream that can be read with a new
 * decompression anddeserialization stream.
 * <p>
 * Unlike {@link org.apache.spark.util.collection.ExternalSorter},this sorter does not merge
 * its spill files. Instead,this merging is performed in {@linkUnsafeShuffleWriter},which
 * uses a specialized merge procedure that avoids extra serialization/deserialization.
 *
 * 为sort-base Shuffle定制的外部排序器
 * 输入的记录会附加到数据pages中。当所有的数据插入后(或当前线程分配的
 * Shuffle内存达到极限时),内存中(in-memory)的记录会根据分区ID进行排序(使
 * 用ShuffleInMemorySorter)。排序后的记录会写入一个输出文件(或多个文件,如
 * 果发生spilled的话)。输出文件的格式与SortShuffleWriter的输出文件个数相同:
 * 每个输出的分区记录通过一个序列化的、压缩的流写入。之后再使用解压的、反序列
 * 化的流进行读取
 * 与ExternalSorter不同,ExternalSorter排序器不会进行spill文件的合并。
 * 而ShuffleExternalSorter排序器在最后会进行合并,使用一个可以避免序列化/反序列
 * 化的特定的合并过程
 * ShuffleExternalSorter继承了MemoryConsumer,会向TaskMemoryManager申请/释
 * 放Execution内存
 */
final class ShuffleExternalSorter extendsMemoryConsumer {
```

可以看到，ShuffleExternalSorter继承了MemoryConsumer，因此在数据处理时可以向TaskMemoryManager申请/释放Execution内存。

对应的，其他建立在 Project Tungsten 内存模型基础上的数据处理也可以通过查看 MemoryConsumer 的子类来获取，比如前面 SPARK－7080 提到的设计文档中的 BytesToBytesMap 子类。

在详细解析源代码之前，同样先给出 ShuffleExternalSorter 在内存处理上的流程，如图 8-9所示。

步骤如下。

1）首先是插入记录，由 UnsafeShuffleWriter 调用 ShuffleExternalSorter 的 insertRecord 方法，向 currentPage 插入一条记录，即图 8-9 中的第 1 步 insertRecord。

2）此时，如果当前页内存未分配，或剩余空间不足以容纳记录数据，则向 TaskMemoryManager 申请内存，即图 8-9 中的第 2 步 allocatePage。

3）在申请内存时，有可能由于内存压力而发出 MemoryConsumer（即这里的 ShuffleExternalSorter）的 spill 操作，即图 8-9 中的第 3 步 spill。

4）触发 spill 操作时，会获取 ShuffleInMemorySorter 的排序数据的迭代器，将排序后的数据 spill 到文件中，即图 8-9 中的第 8 步 getSortedIterator；同时再会释放 ShuffleExternalSorter 占用的内存（通过由记录的内存页 allocatedPages 实现），即图 8-9 中的第 4 步 freePage。

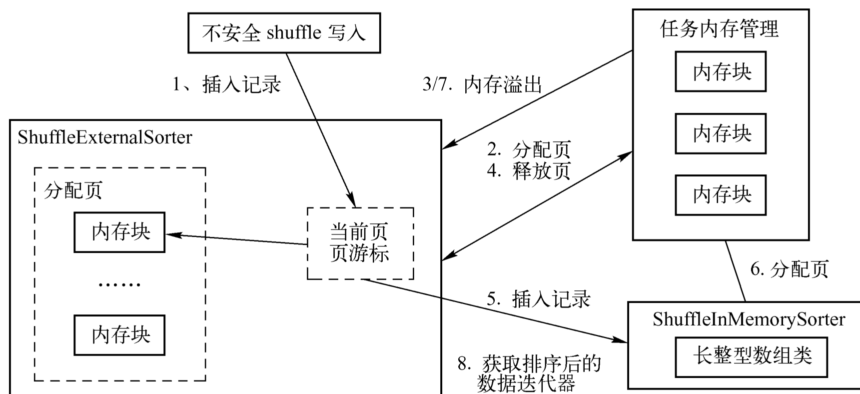


图 8-9　ShuffleExternalSorter 处理流程图

5）当 currentPage 能够容纳记录数据时，将数据插入到内存页中，同时会将记录的编码地址插入到 ShuffleInMemorySorter 的 LongArray 中，即图 8-9 中的第 5 步 insertRecord。

6）插入编码的地址过程中，也可能会由于 LongArray 内存不足而向 TaskMemoryManager 申请内存，即图 8-9 中的第 6 步 allocatePage。申请过程中也可能会触发 spill，这和前面内存申请时描述的过程一样。

7）最后，在完成记录插入后也会调用第 8 步的 getSortedIterator，获取在 ShuffleInMemorySorter 的 LongArray 中未 spill 到文件的内存数据，然后写入最后一个文件（所以这个文件写入的数据量不作为 spill 的 Metric 度量信息）。

在整个处理流程中，比较难以理解的是数据或地址在内存中的存储与处理（实际是用与存储相反的过程来读取数据进行处理，所以本质上还是理解数据是以何种方式存入内存页的）。

8.3.1 插入记录时二进制数据的处理

ShuffleExternalSorter 类的内部处理过程，可以先从该类在 UnsafeShuffleWriter 中的使用开始去理解。如果已经熟悉了 UnsafeShuffleWriter 的大致的写数据过程，可以直接忽略；如果不熟悉，也可以简单地在 UnsafeShuffleWriter 类中搜索 ShuffleExternalSorter 类的调用点来获取二进制处理的入口。

下面直接从 ShuffleExternalSorter 类基于 Project Tungsten 内存模型处理数据的关键入口点开始解析插入记录（对应 ShuffleExternalSorter 的在 UnsafeShuffleWriter 类的 open 与 stop 方法中的处理可以暂时忽略），也就是图 8-9 中的第 1 步插入记录（insertRecord）。即从 UnsafeShuffleWriter 的 insertRecordIntoSorter 方法中的相关源代码部分开始，简单源代码如下。

```
1.    void insertRecordIntoSorter(Product2<K,V> record) throwsIOException {
2.  …
3.      sorter.insertRecord(
4.      serBuffer.getBuf(), Platform.BYTE_ARRAY_OFFSET, serializedRecordSize, partitionId);
5.    }
```

其中第 3 ～ 5 行是解析的关键入口点，也就是 ShuffleExternalSorter 的 insertRecord 方法。在该方法中会把当前的 serBuffer 内容（即一条记录数据）插入到 ShuffleExternalSorter 中。因此，接下来首先分析 insertRecord 方法，对应源代码及其解析如下。

```
  /**
   * Write a record to the shuffle sorter.
   * 向 Shuffle 排序器写入一条记录
   */
  public void insertRecord(Object recordBase, long recordOffset, int length, int partitionId)
    throwsIOException {
…

    //如果需要的话,增加内存中排序所需的 LongArray 内存大小
    growPointerArrayIfNecessary();
    // Need 4 bytes to store the record length.
    //记录插入时,以记录长度作为起始信息,然后是对应该长度的记录数据
    //因此申请的内存大小需要考虑长度本身所占的 4 个字节
    final int required = length + 4;
    acquireNewPageIfNecessary(required);

    assert(currentPage != null);
    //获取当前页的 base object
    final Object base = currentPage.getBaseObject();

    //针对 currentPage 内的页游标(当前起始地址)进行编码
    //该地址对应该记录的起始地址(格式:4 字节的长度 + 记录数据)
```

```
        final longrecordAddress = taskMemoryManager. encodePageNumberAndOffset(currentPage,
        pageCursor);

       //首先将记录的长度(Int,4 字节)放入 base + pageCursor 对应的内存地址,
       //更新当前页内地址 pageCursor
       Platform. putInt(base,pageCursor,length);
       pageCursor += 4;

       //将记录数据 recordBase + recordOffset 复制到 base + pageCursor(已更新 4 字节)
       //中,长度为记录长度,更新当前页内地址 pageCursor
       Platform. copyMemory(recordBase,recordOffset,base,pageCursor,length);
       pageCursor += length;

       //将记录的编码地址(PageNumber + 页内 Offset)及其分区号插入到
       ///内存排序器中
       inMemSorter. insertRecord(recordAddress,partitionId);
    }
```

处理的大致流程在前面已经给出简单描述，这里主要分析记录是如何存储在内存页（Page）的，即内存页中记录的组织形式，可以通过图 8-10 来描述。

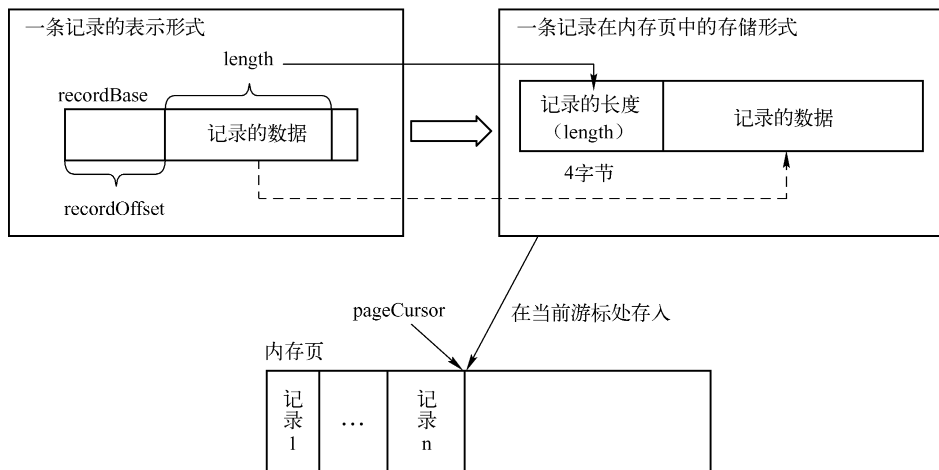


图 8-10　内存页中记录的组织形式

将记录存入内存页时，首先是在内存页当前游标（pageCursor）所在位置存放该记录的数据长度（长度 long 类型对应 4 字节，因此占用的空间是记录数据的长度 +4 字节），然后通过描述记录数据信息的三元素，将记录数据复制到数据长度后面的内存页空间中。

描述记录数据信息的三元素如下。

1）recordBase：记录所在的对象。

2）recordOffset：记录在对象中的偏移量。

3）length：记录数据的长度。

在处理过程中，通过 TaskMemoryManager 的 encodePageNumberAndOffset 方法，将记录在内存页中的存储地址进行编码（存储时已经包含记录的长度和数据，因此只需要起始地址即可），并将该编码地址和所在分区 ID 一起插入到 inMemSorter（ShuffleInMemorySorter）变量中。继续查看插入的信息是如何在 ShuffleInMemorySorter 中组织的，具体源代码如下。

```
public void insertRecord(long recordPointer,int partitionId){
    if(! hasSpaceForAnotherRecord()){
      expandPointerArray(consumer.allocateArray(array.size() *2));
    }

    //以 PackedRecordPointer 封装记录数据的编码地址,
    //编码格式为:24bit - PartitionId + 13bit - PageNumber + 27bit - offset
    //说明:array 对应的元素为 Long 类型,因此长度为 64bit
    array.set(pos,PackedRecordPointer.packPointer(recordPointer,partitionId));
    pos ++;
  }
```

在插入到 ShuffleInMemorySorter 时，会将信息重新封装为 PackedRecordPointer，然后存放到 LongArray 中。下面首先给出一个示意图，如图 8-11 所示。

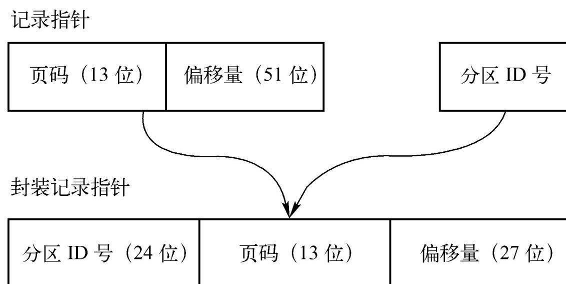


图 8-11　PackedRecordPointer 封装示意图

最终将记录的编码地址 recordPointer 通过 PackedRecordPointer 包装成一个 64 位的 long 型地址。即在 ShuffleInMemorySorter 的 LongArray 中存放的是重新包装后的地址，从该地址可以看到，地址中包含了分区 ID 信息 PartitionId，该信息是用于记录排序的。对应的页内偏移量也缩成了 27 位，因此在使用 Project Tungsten 内存模型时，记录的长度也从原先的 $2^{31}-1$ 变成了 $2^{27}-1$（即当记录长度超过该值时，无法使用基于 Tungsten 的 Shuffle 机制）。同样，使用时分区的数量也会受到限制，即只能有 $2^{24}-1$ 个分区。

对应在插入数据的过程中，使用的两个方法在某些细节上可能不容易理解，因此这里也给出简单的源代码及其解析。

首先是 growPointerArrayIfNecessary 方法，具体代码如下。

```
  private void growPointerArrayIfNecessary() throwsIOException {
    assert(inMemSorter ! = null);
    if(! inMemSorter.hasSpaceForAnotherRecord()){
```



```
        long used = inMemSorter.getMemoryUsage();
        LongArray array;
        try {
          // could trigger spilling
          //内部会通过 TaskMemoryManager 的 allocatePage 方法申请内存,
          //在申请时遇到内存不足会采用一定的策略进行 spill
          array = allocateArray(used / 8 * 2);
        } catch(OutOfMemoryError e) {
          // should have trigger spilling
          assert(inMemSorter.hasSpaceForAnotherRecord());
          return;
        }
        // check if spilling is triggered or not
        //如果在申请内存过程中触发了 spill,使得 inMemSorter 有空间容纳另一条记录,
        //则释放刚申请的 array (LongArray 内部组合了申请的内存块 MemoryBlock)
        if(inMemSorter.hasSpaceForAnotherRecord()) {
          freeArray(array);
        } else {
          //如果没有触发 spill,则 inMemSorter 使用新的 array(内部会有旧数据的迁移)
          inMemSorter.expandPointerArray(array);
        }
      }
    }
```

其次是 acquireNewPageIfNecessary 方法，具体代码如下。

```
  /** 为了插入一些新的记录而申请更多的内存
    * 会从内存管理处申请所需的内存,并在申请失败时触发 spill
    */
    private void acquireNewPageIfNecessary(int required) {
      //当初始情况或当前页(currentPage)剩余空间不足以容易所要求的大小时,
      //申请新的内存页,并更新当前页的游标(指向当前页中可用内存的起始位置),
      //同时将当前内存也放入 allocatedPages,以便后续 spill 或 stop 等情况下释放全部页
      内存
      if(currentPage == null ||
        pageCursor + required > currentPage.getBaseOffset() + currentPage.size()) {
        // TODO:try to find space in previous pages
        currentPage = allocatePage(required);
        pageCursor = currentPage.getBaseOffset();
        allocatedPages.add(currentPage);
      }
    }
```

8.3.2 spill 时二进制数据的处理

可以将继承 MemoryConsumer 类的子类作为二进制解析数据的关键入口点，从前面对 TaskMemoryManager 类的解析源代码中已经知道，当内存不足时，会采用某种策略调用 MemoryConsumer 的 spill 方法（对应策略比较简单，可以直接参考源代码），因此，spill 方法也是理解 ShuffleExternalSorter 类内部处理过程的一个关键入口点。也就是图 8-9 中的第 3 步和第 7 步 spill。

下面是 spill 的关键代码及其解析。

```
/ **
 * Sort and spill the current records in response to memory pressure.
 * 在内存压力下,排序并 spill 当前记录集
 */
@ Override
public long spill(long size,MemoryConsumer trigger)throws IOException {
  …
  writeSortedFile(false);

  //writeSortedFile 不会释放内存,因此需要手动释放
  final longspillSize = freeMemory();
  taskContext. taskMetrics(). incMemoryBytesSpilled(spillSize);
  return spillSize;
}
```

在 spill 时就会调用 writeSortedFile 方法（记录集处理完成后也会调用，只是参数不同），之后便释放占用的全部内存。对应 writeSortedFile 方法的解析，在理解了记录在内存页中存储的组织形式之后，相对比较好理解，因此这里仅给出该方法的注释信息，具体代码如下。

```
/ **
 * Sorts the in - memory records and writes the sorted records to an on - disk file.
 * This method does not free the sort data structures.
 *
 * @ param isLastFile if true,this indicates that we' re writing the final output file and that
 *     the bytes written should be counted towards shuffle spill metrics
 *     rather than   shuffle write metrics.
 *
 * 在内存中对记录进行排序,并写入磁盘的一个文件中
 * 该方法并不会释放排序的数据结构(即需要手动释放内存)
 * 当 isLastFile 为:
 *   1. true 时,表示最后一个输出文件,此时写的数据量统计在 Shuffle write 的度量信息中
 *   2. false 时,则同时统计在 Shuffle write 和 Shuffle spill 的度量信息中
 */
private void writeSortedFile(boolean isLastFile)throws IOException {
```

至此，基本上解析了基于 Project Tungsten 内存模型的整个 Shuffle 数据处理过程，作为扩展，下面简单描述一下内存中的排序（对应 ShuffleInMemorySorter 类），排序时主要使用了 TimSort 排序算法（封装 Java 上的实现），所采用的比较器为 SortComparator，对应的定义如下。

```
private static final classSortComparator implements Comparator < PackedRecordPointer > {
   @ Override
   public int compare( PackedRecordPointer left,PackedRecordPointer right) {
      return left. getPartitionId( ) - right. getPartitionId( );
   }
}
```

从源代码中可以看到，比较时，使用的是 PackedRecordPointer 对象中的分区 ID，所以在基于 Tungsten 的 Shuffle 机制中，记录是按分区 ID 进行排序，并没有对分区内部的记录进行排序。

基于 Project Tungsten 内存模型的数据处理，从 MemoryConsumer 角度出发（包含那些内部使用了 MemoryConsumer 的类，如 UnsafeExternalSorter），逐个去理解内部的数据结构组织与二进制数据处理等方式。

说明：通常在代码中，有 open 方法的话，也会同时对应有 stop 方法，就像资源有申请同时也会有释放，或者元数据信息有创建也会有销毁一样。这些配对的处理方式可以在阅读源代码时注意一下，笔者在源代码解析时可能不会过于强调这点，但实际上这点是很重要的。

Section 8.4 小结

本章内容的主旨在于抛砖引玉，在结合 Databricks 公司发布的博文和 Spark issues 及其相关的设计文档的基础上，从源代码角度对目前已经实现的部分 Project Tungsten 进行解析，主要内容包含 Project Tungsten 的内存模型，以及在该内存模型基础上，以 Shuffle 写数据的过程为例，详细解析在 Project Tungsten 的内存模型上对数据结构进行组织，以及基于二进制处理数据等方面的内容。

第 9 章　性能优化

Spark 自诞生以来，高效极致的运算性能就一直是其不懈的追求，如何写出高效的程序，合理地配置参数，并针对执行过程中的问题做出性能诊断和优化，也是 Spark 高手必须掌握的知识。本章首先讲解 Spark 的配置机制，然后讲解如何通过查看程序运行时的细节信息来进行性能诊断，最后讲解常见的 spark 性能优化的方法。

Section 9.1 Spark 的配置机制

Spark 默认的配置是为了能让 Spark 在大多数集群，包括那些配置比较低的集群上运行起来，所以显然默认的配置不一定适合特定用户的集群，不一定能够最大化地使用集群资源，因此需要针对自己的集群状况，对一些配置做出优化调整。

9.1.1 通过 SparkConf 配置 Spark

常见的配置方式是在程序中通过 SparkConf 来进行配置。SparkConf 包含一系列 key/value 对用来覆盖默认的配置。在使用时，只需要实例化 SparkConf，然后调用它的 set( ) 方法来设置具体配置项。

一个简单的实例代码如下。

```
object WordCount {
  def main(args:Array[String]) {
    //实例化 SparkConf
    val conf = new SparkConf()
    //设置程序名称
    conf.set("spark.app.name","My Spark App")
    //设置 master
    conf.set("spark.master","local[4]")
    //设置 port
    conf.set("spark.ui.port","36000")
    //基于 SparkConf 实例化 SparkContext
    val sc = new SparkContext(conf)
    //具体的业务逻辑
    …
  }
}
```

除了 set( )方法外，SparkConf 针对常用的配置项也专门提供了对应的方法，如可以用 setAppName( )和 setMaster( )来配置 spark. app. name 和 spark. master，所以上述代码也可以写成下面这种形式。

```
object WordCount {
  def main(args:Array[String]){
    //实例化 SparkConf
    val conf = new SparkConf( )
    //设置程序名称
    conf. setAppName("My Spark App")
    //设置 master
    conf. setMaster(local[4])
    //设置 port
    conf. set("spark. ui. port","36000")
    //基于 SparkConf 实例化 SparkContext
    val sc = new SparkContext(conf)
    //具体的业务逻辑
    …
  }
}
```

9.1.2 通过 spark – submit 配置 Spark

通过 SparkConf 配置 spark 是通过在程序中硬编码来配置的，而在大多数情况下，针对具体的应用程序动态地进行配置更为灵活方便。Spark 提供了一种机制，允许用户在使用 spark – submit 提交程序时动态地提供配置项，这些配置项能被 spark 识别并在构建 SparkConf 实例时使用。所以，可以在程序中构建空的 SparkConf 实例，并用它来进一步构建 SparkContext，而具体的配置项则可以在用 spark – submit 提交程序时针对具体的集群状态动态地提供。spark – submit 有一个通用的 flag：– conf，通过它可以对任意配置项赋值；除此之外，spark – submit 针对常见的配置参数专门提供了一系列内置的 flag 以方便使用。

因此，可以将程序代码写成下列形式。

```
object WordCount {
  def main(args:Array[String]){
    //实例化空的 SparkConf
    val conf = new SparkConf( )
    //基于 SparkConf 实例化 SparkContext
    val sc = new SparkContext(conf)
    //具体的业务逻辑
    …
  }
}
```

然后用下面的 spark – submit 在提交程序时指定参数。

```
$ bin/spark – submit \
—class com. example. MyApp \
—master local[4] \
—name "My Spark App" \
—conf spark. ui. port = 36000 \
myApp. jar
```

9.1.3 通过配置文件配置 Spark

spark – submit 也支持从文件中读取配置项，这对于设置多用户多应用共享的环境参数而言非常有用。默认情况下，spark – sbumit 会尝试在 Spark 根目录的 conf 目录下查找文件 spark – defaults. conf，该文件包含空格间隔的一系列 key/value 键值对。当然也可以用 – properties – file 来指定具体的文件路径，如下所示。

```
$ bin/spark – submit \
—class com. example. MyApp \
—properties – file my – config. conf \
myApp. jar
## Contents of my – config. conf ##
spark. master local[4]
spark. app. name "My Spark App"
spark. ui. port 36000
```

9.1.4 Spark 配置机制总结

通过以上分析，总结起来可知，Spark 中的配置项有若干个地方可以配置：通过文件（如 conf/spark – defaults. confh）配置；通过 spark – submit 在提交程序时配置；通过程序代码中的 SparkConf 硬编码配置。有些情况下，同样的配置项可能在这些不同的地方都做了配置，比如在程序中通过 setAppName()设置了程序名，在 spark – submit 中也通过 – name 设置了程序名。那么究竟哪个生效呢？这就涉及优先级问题。

程序硬编码中通过 SparkConf 对象的 set()方法指定的配置项优先级最高，其次是 spark – submit 提交程序时通过 – flag 设置的配置项，然后是参数文件中的配置项，最后是默认值。

要想知道程序运行时配置项的具体值，可以查看程序运行的 webUI。

在 spark – Shell 交互式编程终端下，可以通过 sc. getConf. toDebugString 来查看 SparkConf 的值，通过 sc. hadoopConfiguration. iterator()来获得 Hadoop 的配置信息。

需要说明的一点是，几乎所有的配置都可以通过 SparkConf 来配实现，但是有一个例外：SPARK_LOCAL_DIRS，该参数用来配置 Spark 的本地存储目录（这些目录在 Shuffle 时需要用到）。由于在具体的物理结点上该目录可能不同，需要在 conf/spark – env. sh 中通过 export SPARK_LOCAL_DIRS 到一系列逗号分隔的目录来配置该配置项。

Section
9.2 性能诊断

在进行性能优化前，首先需要了解程序的运行状况，做出性能诊断后，才能有针对性地进行优化。Spark 记录了应用程序详细的进度信息、性能指标等细节信息，这些信息通过两种方式提供给用户使用：①WebUI；②Driver 和 Executor 的日志。可以通过查看分析 WebUI 和日志文件来了解应用程序执行的细节，进行诊断进而找到性能瓶颈，最后做出有针对性的优化。

需要说明的是，WebUI 由若干个 Tab 页面构成，在不同的 Spark 版本中 Tab 页面可能略有不同，这里基于当前最新的 Spark1.6.1 版本进行说明。

9.2.1 WebUI 的 8080 端口

集群 WebUI 的 8080 主页面如下所示，在这里可以查看集群的状态信息，如 Worker 是否正常，Worker 的 Core 和 Memory 大小，正在运行的应用程序，以及最近结束的应用程序的信息等，如图 9-1 所示。

Spark 1.6.1 Spark Master at spark://master:7077

URL: spark://master:7077
REST URL: spark://master:6066 (cluster mode)
Alive Workers: 2
Cores in use: 4 Total, 0 Used
Memory in use: 4.0 GB Total, 0.0 B Used
Applications: 0 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160504024659-192.168.198.144-46799	192.168.198.144:46799	ALIVE	2 (0 Used)	2.0 GB (0.0 B Used)
worker-20160504024705-192.168.198.142-37640	192.168.198.142:37640	ALIVE	2 (0 Used)	2.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160504161057-0001	Spark word count demo	4	2.0 GB	2016/05/04 16:10:57	root	FINISHED	55 s
app-20160504150645-0000	Spark shell	4	2.0 GB	2016/05/04 15:06:45	root	FINISHED	1.0 h

图 9-1 WebUI 的 8080 主页面

在 WebUI 的 8080 主页面单击正在运行的或最近运行结束的应用程序的 Application id，可以进入如图 9-2 所示的页面，这是 Application 的主页面，在这里可以看到该 Application 的相关信息，如是否获得了 Executor，Executor 所在的 Worker 结点，以及 Executor 的 Core、Memory 和 State 等信息，当然也包括 Executor 的日志信息，如图 9-2 所示。

在 WebUI 的 8080 主页面单击正在运行的或最近运行结束的应用程序的 Name，可以进入如图 9-3 所示的页面，这是 Application 的详细情况页面，包括若干个 Tab，即 Jobs、Stages、Storage、Environment 和 Executors，如图 9-3 所示。



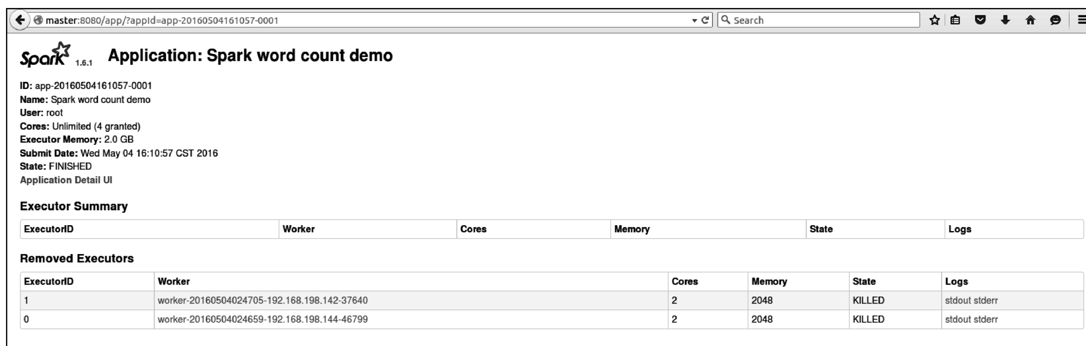


图 9-2　Application 的主页面

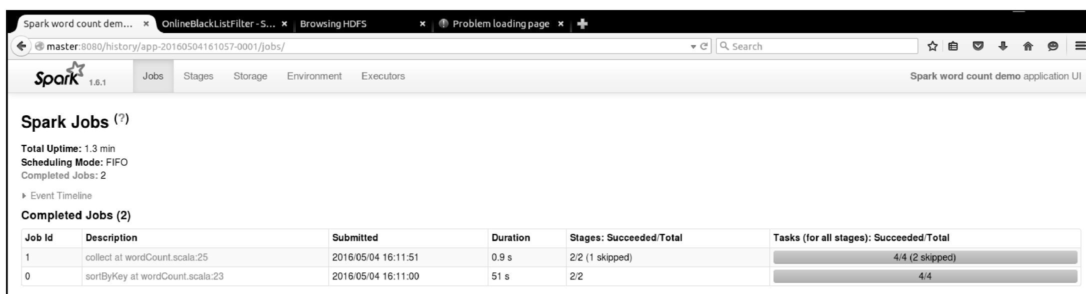


图 9-3　Application 的详细情况页面

9.2.2　WebUI 的 18080 端口

通过集群 WebUI 的 8080 端口能够看到正在运行的和最近运行结束的应用程序的信息，而对于那些较早前运行结束的应用程序，则需要通过 18080 端口来访问 History Server，从而查看集群运行的应用程序的历史信息。

在 History Server 的主页面中，可以查看集群的状态信息，如 worker 是否正常，worker 的 core 和 memory 大小，正在运行的应用程序，以及最近结束的应用程序的信息等，如图 9-4 所示。

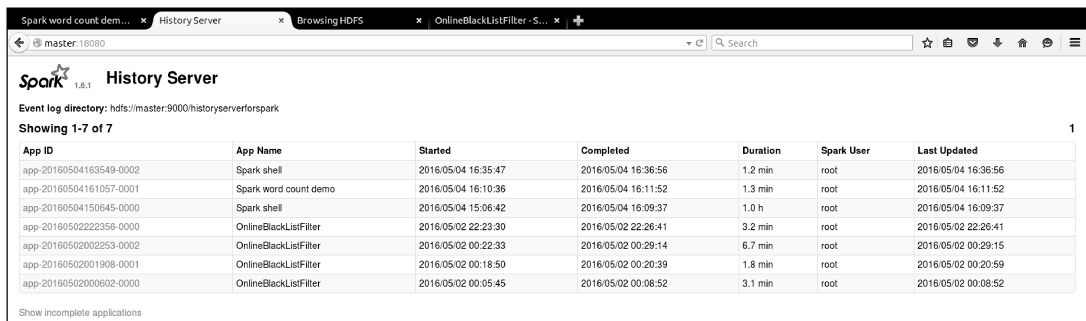


图 9-4　History Server 的主页面

在 History Server 的主页面单击某个应用程序的 app id，即可进入 Application 的详细情况页面，该页面包括若干个 tab，即 Jobs、Stages、Storage、Environment 和 Executors，该页面与通过 8080 端口进入的 Application 的详细情况页面一样，如图 9-5 所示。

图 9-5　History Server 的 Application 详细情况页面

9.2.3　WebUI 的 4040 端口

程序执行过程中也可以通过 4040 端口查看 Application 的详细情况页面，该页面包括若干个 tab，即 Jobs、Stages、Storage、Environment 和 Executors，这跟通过 8080 端口或 18080 端口进入的 Application 的详细情况页面一样，如图 9-6 所示。

图 9-6　4040 端口的 Application 详细情况页面

不管通过哪种方式，进入的 Application 的详细情况页面都包含若干个 Tab 页面，即 Jobs、Stages、Storage、Environment 和 Executors，这些 Tab 页面向用户展示了程序的运行细节，接下来仔细讲解各个 Tab 页面。

9.2.4　WebUI 的 Jobs 页面

Job 页面展示了 job 的详细执行信息，如图 9-7 所示。

单击某个 Job 可以进入该 Job 的细节页面，细节页面展示了该 Job 的 Stages 和 Tasks 的细节，如图 9-8 所示。



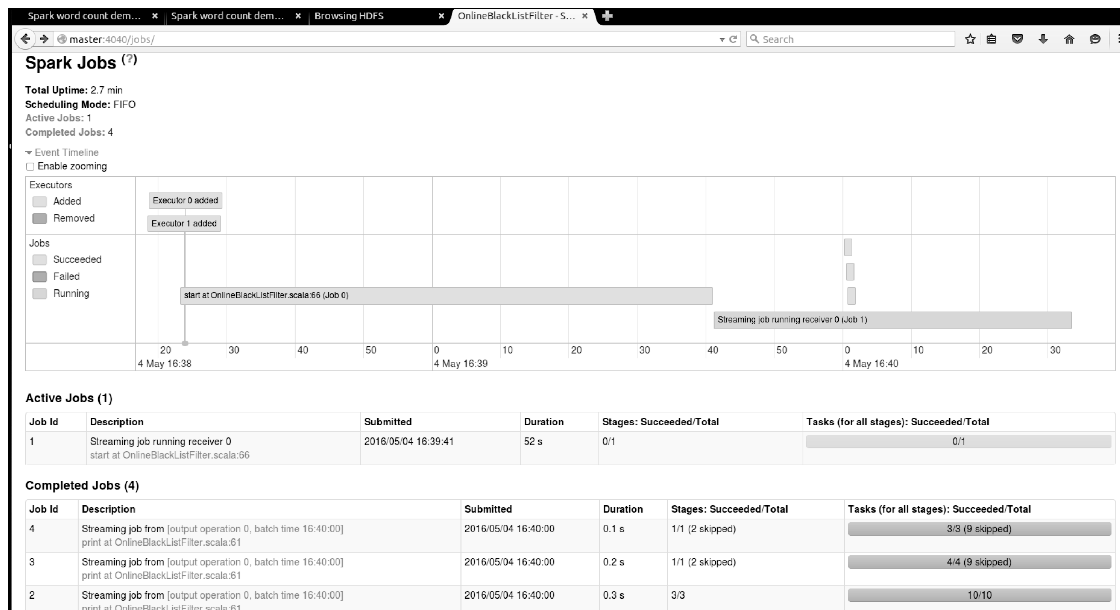


图 9-7　Jobs 主页面

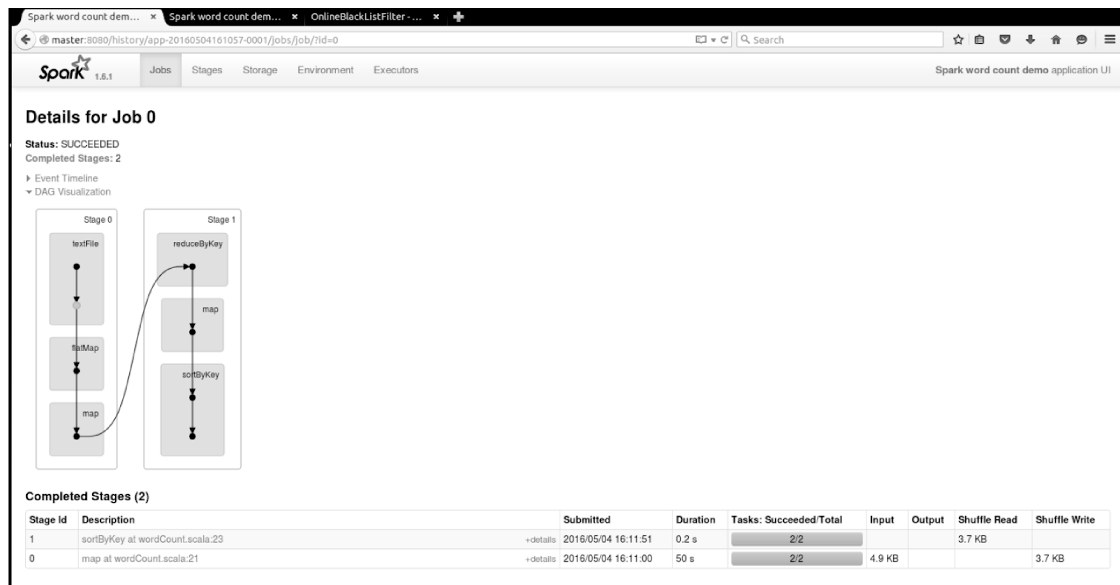


图 9-8　Jobs 的细节页面

9.2.5　WebUI 的 Stages 页面

Stage 页面展示了 Stage 的详细执行信息，在每个 Stage 内部，该页面也提供了若干 metrics 来帮助用户更好地理解作业物理执行的细节。常用该页面来评估 Job 的 performance。

通常首先会查看构成 Job 的各个 Stage，查看是否有运行非常缓慢的 Stage，或在该 Job 的若干次运行时，响应时间变化很大的 Stage。确定好了问题 Stage 后，会进一步查看该

Stage 的详细情况，以定位性能瓶颈。

分布式系统常见的一个性能问题是“倾斜”，即若干个 Task 相对于其他大多数 Task 来说消耗了相当长的时间，可以通过查看 Task 的 Metrics 来判断是否有倾斜。Task 运行了多久，是否有些 Task 相对其他 Task 来说需要多得多的运行时间，如果是的话，就需要进一步分析这些 Task 执行慢的原因；是否有些 Task 相对其他 Task 来说，读或写了多得多的数据；是否某些结点上的 Task 都运行得特别慢，等等。这些都是在进行性能诊断时首先要关注的问题。还要关注 Task 在读，计算和写的各个阶段消耗了多少时间，如果 Task 读写数据消耗的时间不多而整体消耗时间较多，则可能是因为应用程序代码的问题，就需要考虑代码的优化；也可能有些 Task 几乎所有的时间都消耗在从外部存储系统读取数据上了，这时瓶颈在输入的读取上，单纯优化 Spark 可能就没什么很大帮助了。

Stages 页面的主页如图 9-9 所示。

Spark word count dem... × Spark word count dem... × OnlineBlackListFilter - S... ×
master:8080/history/app-20160504161057-0001/stages/
Spark 1.6.1 Jobs Stages Storage Environment Executors Spark word count demo application UI

Stages for All Jobs

Completed Stages: 4

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	collect at wordCount.scala:25 +details	2016/05/04 16:11:51	0.8 s	2/2			3.1 KB	
3	map at wordCount.scala:23 +details	2016/05/04 16:11:51	77 ms	2/2			3.7 KB	3.1 KB
1	sortByKey at wordCount.scala:23 +details	2016/05/04 16:11:51	0.2 s	2/2			3.7 KB	
0	map at wordCount.scala:21 +details	2016/05/04 16:11:00	50 s	2/2	4.9 KB			3.7 KB

图 9-9　Stages 主页面

单击某个 Stage，即可以进入到 Stage 的细节页面，如图 9-10 所示。

Spark word count dem... × Spark word count dem... × OnlineBlackListFilter - S... ×
master:8080/history/app-20160504161057-0001/stages/stage/?id=4&attempt=0
Spark 1.6.1 Jobs Stages Storage Environment Executors Spark word count demo application UI

Details for Stage 4 (Attempt 0)

Total Time Across All Tasks: 63 ms
Locality Level Summary: Node local: 2
Shuffle Read: 3.1 KB / 260

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	31 ms	31 ms	32 ms	32 ms	32 ms
Scheduler Delay	19 ms	19 ms	0.1 s	0.1 s	0.1 s
Task Deserialization Time	0.6 s	0.6 s	0.7 s	0.7 s	0.7 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Result Serialization Time	1 ms	1 ms	1 ms	1 ms	1 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	6.9 KB	6.9 KB	18.1 KB	18.1 KB	18.1 KB
Shuffle Read Blocked Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	911.0 B / 72	911.0 B / 72	2.2 KB / 188	2.2 KB / 188	2.2 KB / 188
Shuffle Remote Reads	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	worker2:52401	2 s	2	0	2	3.1 KB / 260

图 9-10　Stage 的细节页面

9.2.6　WebUI 的 Storage 页面

Storage 页面包含了持久化的 RDD 的相关信息。

大家知道，当我们在应用程序中显式地对一个 RDD 调用了 persist( ) 或 cache( ) 后，在后续通过某个 Job 计算了该 RDD 后，该 RDD 就会被持久化起来。通过对持久化的源代码的分析，可以知道，当对多个 RDD 持久化时，如果存储空间不够，较早的 RDD 可能会被清除，以便为较新的 RDD 腾出空间。

了解程序执行时持久化的细节，是在进行性能诊断时必须关注的。到哪里了解这些细节呢？图 9-11 所示的页面向用户展示了缓存的 RDD 的比例，缓存后 RDD 的大小，缓存的级别（是缓存到磁盘，缓存到内存，还是都有，等等）。通过这些细节，可以知道重要的数据是否被正确地缓存了。

Storage 页面如图 9-11 所示。

Spark word count dem... × Spark word count dem... × Problem loading page ×

master:8080/history/app-20160504161057-0001/storage/

Spark 1.6.1 Jobs Stages Storage Environment Executors Spark word count demo application UI

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
hdfs://master:9000/data/README.md	Memory Deserialized 1x Replicated	2	100%	10.2 KB	0.0 B	0.0 B

图 9-11 Storage 页面

9.2.7 WebUI 的 Environment 页面

通过 9.1 节中对 Spark 配置机制的分析，可以知道有多种配置方式，其中有些是系统管理员做的配置，有些是提交程序时动态指定的配置，还有些是应用程序中硬编码指定的配置，而且这些配置有优先级，有的配置会覆盖其他地方的配置。

程序运行时真正生效的配置是什么呢？Environment 页面就展示了应用程序执行的过程中所使用的实际的具体配置信息，Environment 页面如图 9-12 ～图 9-14 所示，之所以分成 3 个图，是因为该页面信息很多，分为不同的方面。

Spark word count dem... × Spark word count dem... × Problem loading page ×

master:18080/history/app-20160504161057-0001/environment/

Spark 1.6.1 Jobs Stages Storage Environment Executors Spark word count demo application UI

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/java/jdk1.8.0_60/jre
Java Version	1.8.0_60 (Oracle Corporation)
Scala Version	version 2.10.5

Spark Properties

Name	Value
spark.driver.host	192.168.198.136
spark.history.fs.logDirectory	hdfs://master:9000/historyserverforspark
spark.eventLog.enabled	true
spark.driver.port	44509
spark.jars	file:/root/workspace/SparkCoreAps.jar
spark.app.name	Spark word count demo
spark.scheduler.mode	FIFO
spark.driver.memory	2g
spark.executor.id	driver
spark.submit.deployMode	client
spark.master	spark://master:7077
spark.executor.memory	2g
spark.eventLog.dir	hdfs://master:9000/historyserverforspark
spark.externalBlockStore.folderName	spark-83425834-a97c-4006-a6f1-127782750dd4
spark.app.id	app-20160504161057-0001

图 9-12 Environment 的 Runtime Information 和 Spark Properties 页面

如图 9-12 展示了 Runtime Information 和 Spark Properties，图 9-13 展示了 System Properties，图 9-14 展示了 Classpath Entries。

Spark word count dem... × Namenode information ×
master:18080/history/app-20160504161057-0001/environment/ Search

System Properties

Name	Value
java.io.tmpdir	/tmp
line.separator	
path.separator	:
sun.management.compiler	HotSpot 64-Bit Tiered Compilers
SPARK_SUBMIT	true
sun.cpu.endian	little
java.specification.version	1.8
java.vm.specification.name	Java Virtual Machine Specification
java.vendor	Oracle Corporation
java.vm.specification.version	1.8
user.home	/root
file.encoding.pkg	sun.io
sun.nio.ch.bugLevel	
sun.arch.data.model	64
sun.boot.library.path	/usr/lib/java/jdk1.8.0_60/jre/lib/amd64
user.dir	/root/workspace
java.library.path	/usr/java/packages/lib/amd64:/usr/lib64:/lib64:/lib:/usr/lib
sun.cpu.isalist	
sun.desktop	gnome
os.arch	amd64
java.vm.version	25.60-b23
java.endorsed.dirs	/usr/lib/java/jdk1.8.0_60/jre/lib/endorsed
java.runtime.version	1.8.0_60-b27
java.vm.info	mixed mode
java.ext.dirs	/usr/lib/java/jdk1.8.0_60/jre/lib/ext:/usr/java/packages/lib/ext
java.runtime.name	Java(TM) SE Runtime Environment

图 9-13　Environment 的 System Properties 页面

Spark word count dem... × Namenode information ×
master:18080/history/app-20160504161057-0001/environment/ Search

sun.os.patch.level	unknown
java.vm.specification.vendor	Oracle Corporation
user.country	US
sun.jnu.encoding	UTF-8
user.language	en
java.vendor.url	http://java.oracle.com/
java.awt.printerjob	sun.print.PSPrinterJob
java.awt.graphicsenv	sun.awt.X11GraphicsEnvironment
awt.toolkit	sun.awt.X11.XToolkit
os.name	Linux
java.vm.vendor	Oracle Corporation
java.vendor.url.bug	http://bugreport.sun.com/bugreport/
user.name	root
java.vm.name	Java HotSpot(TM) 64-Bit Server VM
sun.java.command	org.apache.spark.deploy.SparkSubmit --master spark://master:7077 --class com.dt.spark.core.wordCount /root/workspace /SparkCoreAps.jar
java.home	/usr/lib/java/jdk1.8.0_60/jre
java.version	1.8.0_60
sun.io.unicode.encoding	UnicodeLittle

Classpath Entries

Resource	Source
/usr/local/spark/spark-1.6.1-bin-hadoop2.6/lib/spark-assembly-1.6.1-hadoop2.6.0.jar	System Classpath
/usr/local/spark/spark-1.6.1-bin-hadoop2.6/lib/datanucleus-api-jdo-3.2.6.jar	System Classpath
/usr/local/spark/spark-1.6.1-bin-hadoop2.6/lib/datanucleus-rdbms-3.2.9.jar	System Classpath
/usr/local/hadoop/hadoop-2.6.0/etc/hadoop/	System Classpath
http://192.168.198.136:53004/jars/SparkCoreAps.jar	Added By User
/usr/local/spark/spark-1.6.1-bin-hadoop2.6/conf/	System Classpath
/usr/local/spark/spark-1.6.1-bin-hadoop2.6/lib/datanucleus-core-3.2.10.jar	System Classpath

图 9-14　Environment 的 Classpath Entries 页面

需要指出的是，该页面也列举了应用程序执行时实际使用的 jars 和 files，这对于诊断程序运行时的依赖缺失等错误很有帮助。

9.2.8 WebUI 的 Executors 页面

Executors 页面展示了应用程序申请到的所有的 Executors 相关信息，如图 9-15 所示。

master:18080/history/app-20160504161057-0001/executors/ Search

Spark 1.6.1 Jobs Stages Storage Environment Executors

Executors (3)

Memory: 10.2 KB Used (3.7 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
0	worker1:41090	0	0.0 B / 1247.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr
1	worker2:52401	2	10.2 KB / 1247.3 MB	0.0 B	0	0	8	8	1.7 m	4.9 KB	0.0 B	6.8 KB	stdout stderr
driver	192.168.198.136:49047	0	0.0 B / 1247.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

图 9-15　Executors 页面

性能诊断时，需要首先查看该页面以确定应用程序是否申请到了足够多的 Executors 资源。由于集群环境的复杂性，很多时候，集群中的某些结点上不恰当的配置会导致在这些结点上申请不到 Executors 来执行应用程序，则应用程序所能申请到的总的 Executors 个数就不一定满足需要，性能相应地就不理想，此时就要调整这些相应结点的配置信息。

举例来说，如果该页面显示的某些 Executors 上执行的任务中有很多执行的非常慢甚至错误的任务的话，就应该认真关注该 Executor 及其所在 Worker 结点的配置，并进行相应的调整，有必要时甚至可以移除这些结点，以避免在这些结点上分配 Executors 来执行任务。

9.2.9 Driver 和 Executor 的日志

除了 WebUI，另一个可以获取到程序执行时具体信息的地方是 Driver 和 Executor 的日志。这些日志包含了程序执行时很多的细节，如代码抛出的警告和异常等，这些信息也有助于性能诊断。

日志文件的地址跟程序运行的模式有关。在 Standalone 模式下，这些日志在 Master 的 WebUI 上可以直接显示出来，其物理位置默认情况下存储在 Worker 结点 Spark 根目录下的 work 目录下；在 Mesos 模式下，可以通过 Mesos masterUI 来访问这些日志，其物理位置存储在 Mesos 的 Slave 的 Work 目录下；在 YARN 模式下，最简单的收集日志的方式是使用 YARN 提供的日志收集工具（运行命令 yarn logs －applicationId ＜app ID＞），不过该工具只有在程序运行结束后才能收集到这些日志信息，在程序运行期间，需要从 ResourceManager UI 进入

具体的结点，然后进入具体的 container 来查看日志。

需要说明的是，也可以配置 Spark 的日志系统 log4j 来更改日志输出的级别。Spark 提供了一个 log4j 的配置文件模板 conf/log4j. properties. template，可以复制该文件到 log4j. properties，然后修改复制后的文件，比如将 logging level 从默认情况下的 INFO 修改为 WARN 或 ERROR 来产生更少的日志输出信息。配置完成后，可以在 spark - submit 提交程序时通过 - files log4j. properties 来指定程序运行时使用的日志配置文件。

Section 9.3 性能优化

了解了 Spark 的配置机制，也知道了如何查看程序运行时的细节信息来进行性能诊断后，接下来就能有针对性地对性能瓶颈做出优化了。

需要说明的是，如果集群的资源是弹性的，简单地通过增大 CPU 内核数量、增大内存或网络带宽等方式，当然也能提升程序的性能。这里说的性能优化是指在硬件资源固定，也就是集群的内存、CPU、磁盘和网络带宽等资源固定的情况下，通过修改参数，调整配置，甚至修改程序业务逻辑代码，来优化性能。

当然，性能的优化是一个动态的过程，不可能一劳永逸，这里讲解的方法也并没有涵盖所有的内容，只是讲解了常见的 Spark 性能优化的方法，读者在使用过程中需要反复调试、不断摸索，才能找到最适合自己的集群和程序的方案。

9.3.1 程序编写准则

从根本上讲，Spark 程序的性能取决于用户编写的程序。所以在讲解具体的优化方法之前，首先简单描述一下 Spark 程序编写需要注意的地方。在开发过程中，要时刻牢记这些原则，根据具体的业务逻辑，将这些原则结合起来，灵活地运用它们。

1. 准则一：从同一个数据源尽量只创建一个 RDD，后续不同的业务逻辑可以复用该 RDD，而不是基于该数据源重新创建一个新的 RDD

这是因为，从数据源创建 RDD 常常涉及数据的读取，而数据的读取速度一般都比数据计算的速度慢，所以要减少从外部数据源加载数据的次数，尽量复用 RDD 而不是重新创建新的 RDD，这点在程序业务逻辑复杂冗长的情况下常常被忽视，一个错误的示例代码如下所示：

```
object WordCount {
  def main(args:Array[String]) {
    //实例化空的 SparkConf
    val conf = new SparkConf()
    //基于 SparkConf 实例化 SparkContext
    val sc = new SparkContext(conf)
```

```
        //具体的业务逻辑:需要对名为 README.md 的 HDFS 文件进行一次 map 操作,再进行一次 reduce 操作。也就是说,需要对一份数据执行两次算子操作
        // 错误的做法:对同一份数据源执行多次算子操作时,创建了两个 RDD,然后分别对每个 RDD 都执行了一个算子操作。这种情况下,Spark 需要两次从 HDFS 上加载 README.md 文件的内容,并创建两个单独的 RDD。第二次加载 HDFS 文件及创建 RDD 的动作显然是不必要的
        val rdd1 = sc.textFile("hdfs://master:9000/data/README.md")
        rdd1.map(...)
        val rdd2 = sc.textFile("hdfs://master:9000/data/README.md")
        rdd2.reduce(...)
        //进一步的具体业务逻辑
        ...
        // 正确的用法:对同一份数据源执行多次算子操作时,只创建一个 RDD,然后复用该 RDD 执行多次算子操作。这种情况下,Spark 仅仅需要从 HDFS 上加载一次 README.md 文件的内容
        val rdd1 = sc.textFile("hdfs://master:9000/data/README.md")
        rdd1.map(...)
        rdd1.reduce(...)
    }
}
```

当然，由于 Spark 程序执行时的延迟执行和基于 Lineage 最大化的 pipeline 的特性，由于 rdd1 被执行了两次算子操作（一次 map，一次 reduce），在执行 reduce 操作时，还会再次从源头重新计算一次 rdd1 的数据，即再次重新从 HDFS 加载外部数据源的数据。所以，要想彻底解决重复读取外部数据源的问题，还需结合“准则二：如果需要对某个 RDD 进行多次不同的 transformation 和 action 操作以应用于不同的业务分析需求，可以考虑对该 RDD 进行持久化操作，以避免 action 操作触发作业时多次重复计算该 RDD”，才能保证一个 RDD 被多次使用时只被计算一次。

2. 准则二：如果需要对某个 RDD 进行多次不同的 Transformation 和 Action 操作以应用于不同的业务分析需求，可以考虑对该 RDD 进行持久化操作，以避免 Action 操作触发作业时多次重复计算该 RDD

需要考虑该准则是因为 Spark 程序执行的特性，即延迟执行和基于 Lineage 最大化的 pipeline。简单来说，Spark 中由于对某个 RDD 的 Action 操作触发了作业时，会基于 Lineage 从后往前推，找到该 RDD 源头的 RDD，然后从前往后计算出结果。很明显，如果对某个 RDD 执行了多次 Transformation 和 Action 操作，每次 Action 操作触发了作业时都会重新从源头 RDD 处计算一遍来获得该 RDD，然后再对这个 RDD 执行相应的操作。这种方式的性能显然是很差的。

所以需要对多次使用的 RDD 进行持久化。持久化之后，Spark 就会根据持久化策略，将 RDD 中的数据保存到内存或者磁盘中。以后每次对该 RDD 进行算子操作时，都会直接从内存或磁盘中提取持久化的 RDD 数据，然后执行算子，而不会从源头处重新计算一遍这个 RDD，再执行算子操作。

3. 准则三：从数据源读取数据获得 RDD 后，要尽早进行 filter 过滤掉不需要的数据

Spark 是居于内存的迭代计算模型，将不必要的数据尽早过滤掉（filter），减少内存的占用，从而提高程序的执行效率。

需要说明的是，如果 filter 后会获得大量小文件，可能需要通过 repartition 或 coalesce 减小并行度。

4. 准则四：尽量避免使用需要 Shuffle 的算子，且在必须 Shuffle 时尽量减少 shuffle 的数据量

如果有可能，要尽量避免使用 Shuffle 类算子。因为 Spark 作业运行过程中，最消耗性能的地方就是 Shuffle 过程。Shuffle 过程就是数据洗牌，简单来说，就是将分布在集群中多个结点上的包含同一个 key 的数据，拉取到同一个结点上，然后进行聚合或 join 等操作。reduceByKey、join 等算子都会触发 Shuffle 操作。

Shuffle 过程中，各个结点上的相同 key 都会先写入本地磁盘文件中，然后其他结点需要通过网络传输拉取各个结点上的磁盘文件中的含有相同 key 的记录。在将这些含有相同 key 的数据都拉取到同一个结点进行聚合操作时，还有可能会因为一个结点上处理的 key 过多，导致内存不够存放，进而溢写到磁盘文件中。因此在 Shuffle 过程中，可能会发生大量的磁盘文件读写操作，以及数据的网络传输操作，而这无疑也会降低程序的执行速度。

因此在开发过程中，应当尽可能避免使用 reduceByKey、join、distinct 和 repartition 等会进行 Shuffle 的算子，而尽量使用 Map 类的非 Shuffle 算子。这样的话，没有 Shuffle 操作或者仅有较少 Shuffle 操作，程序的执行就可以大大减少性能开销。

如果因为业务需要，一定要使用 Shuffle 操作且无法用 Map 类的算子来替代时，就要尽量减少 Shuffle 的数据量，可以通过使用 map – side 预聚合的算子来减少 Shuffle 的数据量。

所谓的 map – side 预聚合，指的是在每个结点本地 map 时，对含有相同的 key 记录进行了聚合操作，这点类似于 MapReduce 中的本地 combiner。map – side 预聚合之后，每个结点本地就只会有一条含有相同的 key 的记录（因为多条含有相同的 key 的记录都被聚合起来）。其他结点在拉取所有结点上的含有相同的 key 的记录时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘 I/O 和网络传输开销。

这里以 ReduceByKey/AggregateByKey 与 groupByKey 的对比来说明 map – side 预聚合的重要性。ReduceByKey/aggregateByKey 算子会使用用户自定义的函数对每个结点本地的含有相同 key 的记录进行预聚合。而 groupByKey 算子是不会进行预聚合的，全部的数据都会在集群的各个结点之间分发和传输，性能相对来说肯定比较差。

GroupByKey 与 ReduceByKey 的原理图如图 9–16 和图 9–17 所示。

从图中我们可以看出，groupByKey 不会进行 map 端的预聚合，而是将所有 map 端的数据 shuffle 到 reduce 端，然后在 reduce 端进行聚合操作。

从图中我们可以看出，reduceByKey 会首先在 map 端进行预聚合操作，然后将聚合后的数据 shuffle 到 reduce 端，由于 map 端的预聚合操作会减小数据量，所以需要在网络中传输的数据量就减小了，效率也就比 groupByKey 更高。

使用 ReduceByKey 替代 groupByKey 的实例代码如下。

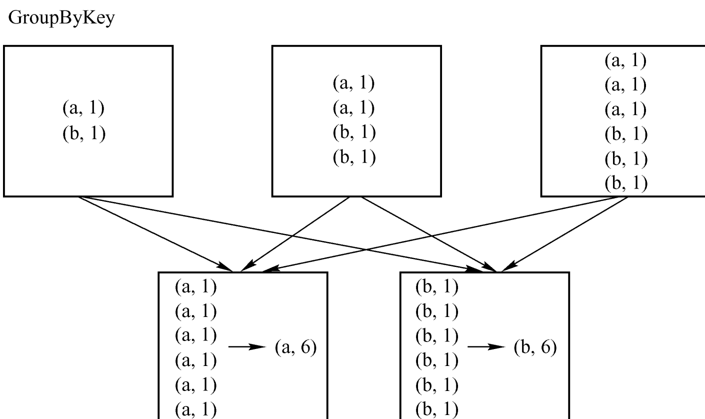


图 9-16　GroupByKey 原理图

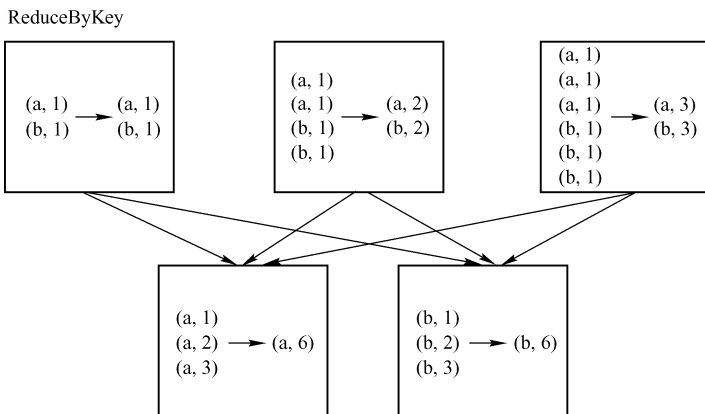


图 9-17　ReduceByKey 原理图

```
val words = Array("Spark","Scala","Hadoop","Java","Mapreduce","RDD")
val wordPairsRDD = sparkcontext.parallelize(words).map(word => (word,1))
//使用 groupByKey 的实现
val WordCountByGroup = wordPairsRDD.groupByKey().map(t => (t._1,t._2.sum)).collect()
//使用 reduceByKey 的实现
val WordCountByReduce = wordPairsRDD.reduceByKey(_ + _).collect()
```

5. 准则五：熟悉各个算子的背后机制，选择使用高性能的算子

Spark 的一大特性就是提高了丰富的算子以满足不同的业务需要。一个良好的程序员需要熟知这些算子的背后机制，能够选择使用高性能的算子满足业务需求。这里列举几个常见的实例。

相对来说，可以使用 reduceByKey/aggregateByKey 替代 groupByKey。

相对来说，可以使用 mapPartitions 替代普通 Map，因为 mapPartitions 类的算子，一次函数调用会处理一个 Partition 所有的数据，而不是一次函数调用处理一条，所有性能相对来说会高一些。但是有时候，使用 mapPartitions 会出现 OOM（内存溢出）的问题。因为单次函数调用就要处理一个 Partition 所有的数据，如果内存不够，垃圾回收时是无法回收太多对象的，很可能出现 OOM 异常。所以使用这类操作时要慎重。

相对来说，可以使用 foreachPartitions 替代 foreach，原理类似于“使用 mapPartitions 替代

Map”，也是一次函数调用处理一个 Partition 的所有数据，而不是一次函数调用处理一条数据。这个实例一个常见的经典应用场景是在写记录到数据库时，如果是普通的 foreach 算子，每次函数调用都需要创建一个数据库连接，然后写一条数据，此时势必会频繁地创建和销毁数据库连接，性能非常低下；但是如果用 foreachPartitions 算子一次性处理一个 Partition 的数据，那么对于每个 Partition 只要创建一个数据库连接，然后执行批量插入操作，此时性能肯定是比较高的。

相对来说，可以使用 repartitionAndSortWithinPartitions 替代 Repartition 与 Sort 类操作。事实上，repartitionAndSortWithinPartitions 是 Spark 官网推荐的一个算子，如果在 Repartition 重分区之后还要进行排序，官方建议直接使用 repartitionAndSortWithinPartitions 算子，因为该算子可以一边进行重分区的 Shuffle 操作，一边进行排序。Shuffle 与 Sort 两个操作同时进行，比先 Shuffle 再 Sort 来说，性能肯定是比较高的。

对一个 RDD 执行 filter 算子后，如果可能过滤掉 RDD 中较多的数据（比如 30% 以上的数据），就建议使用 coalesce 算子，手动减少 RDD 的 Partition 数量，将 RDD 中的数据压缩到更少的 Partition 中去，以减少并行度，避免过多开辟 Task 的开销。因为 Filter 之后，RDD 的每个 Partition 中都会有很多数据被过滤掉，此时如果照常进行后续的计算，每个 Task 处理的 Partition 中的数据量并不是很多，而且此时开辟的 Task 越多，可能速度反而越慢，因为 Task 的开辟和销毁也是有开销的。因此可以用 coalesce 来减少 partition 数量，将 RDD 中的数据压缩到更少的 Partition，只需使用更少的 Task 即可处理完所有的 Partition。

6. 准则六：对大变量考虑使用广播机制

有时在开发过程中，会遇到需要在算子函数中使用外部大变量的场景（比如 100M 的大集合），那么此时就可以考虑使用 Spark 的广播（Broadcast）功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark 会将该变量复制多个副本，通过网络传输到 Task 中，此时每个 Task 都有一个变量副本。如果变量本身比较大的话（比如 100 MB，甚至 1 GB），那么大量的变量副本在网络中传输的性能开销，以及在各个结点的 Executor 中占用过多内存导致的频繁 GC，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用 Spark 的广播功能，对该变量进行广播。广播后的变量会保证每个 Executor 的内存中只驻留一份变量副本，而 Executor 中的 Task 执行时共享该 Executor 中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对 Executor 内存的占用开销，降低 GC 的频率。

7. 准则七：尽可能使用 Kryo 优化序列化性能

Spark 默认的序列化器是 org. apache. spark. serializer. JavaSerializer，但同时也支持使用 Kryo 序列化器 org. apache. spark. serializer. KryoSerializer。由于默认的序列化器的性能和空间表现都比较差，而 Kryo 序列化器更快，压缩率也更高，所以我们应该优先使用 Kryo 序列化器而不是默认的序列化器。

8. 准则八：使用优化的数据结构

Spark 应用程序同普通程序一样需要考虑数据结构问题，由于 Spark 是优先基于内存的迭代计算模型，尤其需要考虑使用内存友好的数据结构。在可能及合适的情况下，应该使用占用内存较少的数据结构。

由于数据结构的知识是通用的，不单单只在 Spark 程序里需要考虑，这里仅简单描述一

下常见的数据结构，更详细的情况请读者自行分析。

Java 语言的特性导致有些数据结构会占用额外的空间，在 Java 中有以下 3 种类型的数据结构比较耗费内存。

1）Java 的对象：每个 Java 对象都有对象头，对象头占用额外的 16 个字节（包含指向对象的指针等元数据信息），如果对象中只有一个 int 类型的变量，则此时会占据 20 个字节，也就是说对象的元数据占用了大部分的空间，所以在封装数据时尽量不要使用对象，可以使用 JSON 格式来封装数据。需要注意，Java 中基本的数据类型会自动进行封箱操作，例如，int 会自动变成 Integer，这也会额外增加对象头的空间占用。

2）Java 的字符串：每个字符串内部都有一个字符数组及长度等额外信息，在实际占用内存方面要额外使用 40 个字节（每个字符串内部都使用字符数组来保存字符序列）。由于字符串中的每个字符占用 2 个字节（UTF－16 编码），所以如果字符串内部有 5 个字符的话，实际上会占用 50 个字节。

3）Java 中的集合类型（如 HashMap、List 等）：集合的内部一般使用链表来实现，具体的每个数据则使用 Entry 等，这些也非常消耗内存。

所以 Spark 官方建议用户，在 Spark 编码实现中，特别是对于算子函数中的代码，尽量使用字符串替代对象，使用原始类型（如 Int 和 Long）替代字符串，使用原生数组替代集合类型，这样会尽可能地减少内存占用，从而降低 GC 频率，提升性能。例如，List <Integer> list = new ArrayList <Integer> 需要考虑改为使用 int[ ] arrary = new int[ ]。

如果内存少于 32 GB，可以在 spark－env. sh 中设置 JVM 参数－XX：+UseCompressedOops，以便使用 4 字节指针而不是 8 字节指针。与此同时，在 Java 7 或者更高版本中，设置 JVM 参数－XX：+UseCompressedStrings，以便采用 8 位来编码每一个 ASCII 字符。

9.3.2 并行度

并行度指的就是 RDD 的分区数，由于一个分区对应一个 Task，并行度也是一个 Stage 中的 Task 数，这些 Task 被并行处理。通过前面的章节可以知道，RDD 是以 Partition 即分区的形式散落在集群上的，每个分区都包含一部分待处理的数据，Spark 程序运行时，会为每个待处理的分区创建一个 Task，且默认情况下每个 task 占用一个 CPU Core 来处理。

Spark 有一套自己自动推导出默认的分区数的机制。当在程序中通过操作算子如 textFile 等读取外部数据源以获得 Input RDD 时，Spark 会自动地根据外部数据源的大小推导出一个合适的默认分区数，如 HDFS 文件的每个 Block 就对应一个分区；在对 RDD 进行 Map 类不涉及 Shuffle 的操作时，由于分区数具有遗传性，新产生的 RDD 的分区数由 parent RDD 中最大的分区数决定；在对 RDD 进行 Reduce 类涉及 Shuffle 操作的算子时（如 groupByKey、reduceByKey 等各种 Reduce 操作算子），由于分区数具有遗传性，新产生的 RDD 的分区数也由 Parent RDD 中最大的分区数决定。如果是在 Spark－Shell 交互式命令终端下，可以通过方法 rdd. partitions. size 来获得某个 RDD 的分区数，而在 Spark 1. 6. 0 以后的版本中，也可以通过 rdd. getNumPartitions( ) 来获得某个 RDD 的分区数。

并行度对性能的影响有两方面，当并行度不够大时会存在资源的闲置与浪费，比如一个应用程序分配到了 1000 个 Core，但是一个 stage 里只有 30 个 Task，此时就可以提高并行度以提升硬件利用率；而当并行度太大时，Task 常常几微秒就执行完毕，或 Task 读写的数据

量很小，这种情况下，Task 频繁地开辟与销毁的不必要的开销则太大，就需要调小并行度。

由于 Spark 自动推导出来的默认分区数很多时候并不理想，必须人为地加以控制来改变并行度。Spark 提供了 4 种改变并行度的方式。

第一种，在使用读取外部数据源的 textFile 类算子时，可以通过可选的参数 minPartitions 来显示指定最小的分区数。

第二种，针对已经存在的 RDD，可以通过方法 repartition( ) 或 coalesce( ) 来改变并行度。repartition( ) 和 coalesce( ) 的区别在于，前者会产生 Shuffle，而后者默认不会产生 Shuffle。事实上，当有大量小任务（任务处理的数据量小且耗时短）时，比如某个 RDD 在 Filter 操作后，由于过滤掉了大量数据，每个分区都只剩下了很少量的数据，这时常用 coalesce( ) 来合并分区，调小并行度，减少不必要的任务开辟与销毁的消耗；而当任务耗时长且处理的数据量大时，如果计算只发生在部分 Executor 上，常用 repartition( ) 来重新分区，提高并行度，开辟更多的并行计算的任务来完成计算。

第三种，在对 RDD 进行 Reduce 类涉及 Shuffle 操作的算子时，这些算子大都可以接受一个显式指定的参数来确定新产生的 RDD 的分区数，我们可以显式地指定这类参数来改变 Shuffle 后新产生的 RDD 的分区数，而不是采用系统推导出的默认的分区数。

第四种，也可以配置参数 spark. default. parallelism 来设置默认的并行度。该参数其实指定的就是在对 RDD 进行 Reduce 类涉及 Shuffle 操作的算子时，如果没有对这些算子显示指定参数来确定新产生的 RDD 的分区数时，这类 Reduce 类涉及 Shuffle 操作的算子产生的新的 RDD 的 Paritition 数量。该参数也指定了 Parallelize 等没有 Parent RDDs 类操作的算子所产生的新的 RDD 的分区数。

一个最佳实践是，将并行度设置为集群的总的 CPU Cores 个数的 2 ～ 3 倍，比如 Executor 的总 CPU Core 数量为 400 个，那么设置 1000 个 Task 是可以的，此时可以充分利用 Spark 集群的资源；每个分区的大小在 128 MB 左右。

需要说明的是，通过以上方式确定了任务的并行度，就确定了理论上能够并行执行的任务的数量，而实际执行时真正并发执行的任务数量还要受到应用分配到的实际资源数量的限制，要想改变应用程序获得的资源数目，这就涉及资源参数的调优。

9.3.3 资源参数调优

所谓 Spark 资源参数调优，其实主要就是对 Spark 程序运行过程中各个使用资源的地方，通过调节各种参数来优化资源使用的效率，从而提升 Spark 作业的执行性能。需要考虑的资源主要是分配给各个 Executors 的 Core 的个数和内存，分配给程序的 Executors 总个数，以及分配的本地磁盘数等。

Spark 应用程序的每个 Executors 都有固定的一样的 Core 和内存。

每个 Executors 的内存，在各种模式下，都可以通过 Spark - Submit 的 - executor - memory 参数，或 spark - defaults. conf 配置文件的 spark. executor. memory 参数来指定。Executors 内存的大小很多时候直接决定了 Spark 作业的性能，而且跟常见的 JVM OOM 异常也有直接关联。但该值不是越大越好，因为太大的堆空间会引起 GC 垃圾回收的延迟，该堆大小的最大值建议不超过 64 GB。

每个 Executor 的 Core 个数和申请的 Executors 的总个数，在 YARN 模式下，可以通过 spark - submit 的 - executor - cores 参数和 - num - executors 参数来分别指定，也可以通过

spark - defaults. conf 配置文件中的 spark. executor. cores 参数和 spark. executor. instances 参数来分别指定；在 Standalone 和 Mesos 模式下，由于 Spark 程序会尽可能多地占用所有可用的 Core 和创建尽可能多的 Executors，所以可以通过参数 spark. cores. max 来指定程序申请的 Executors 的 Core 总个数。因为每个 CPU Core 同一时间只能执行一个 Task 线程，所以每个 Executor 进程的 CPU Core 数量越多，越能够快速地执行完分配给自己的所有的 Task 线程。但是还需要考虑到文件系统的 I/O 吞吐量，一个最佳实践是，建议为每个 Executor 设置 5 个或 5 个以下的 Core。

Spark 使用 Executor 的本地磁盘来存储 Shuffle 时的临时数据和 Spill 到磁盘上的被缓存的 RDD 分区，配置使用多个本地磁盘可以提升 Spark 的性能。在 YARN 模式下，YARN 有自己的配置本地磁盘的机制；在 Standalone 模式下，可以在 conf/spark - env. sh 中配置环境变量 SPARK_LOCAL_DIRS；在 Mesos 模式下，可以配置选项 spark. local. dir. 这 3 种配置模式都可以配置多个文件目录，多个文件目录之间需要用逗号隔开。

由于每一台 Host 上面可以并行 N 个 Worker，每一个 Worker 下面可以并行 M 个 Executor，每个 Executor 可以占用一个或多个 Core，而 Task 会被分配到 Executor 上去执行且默认情况下一个 Task 占用一个 Core，所以可以通过观察 CPU 的使用率变化来了解计算资源的使用情况。若 CPU 利用率很低且程序运行缓慢，就可以尝试减少每个 Executor 占用的 cpu cores 的数量，增加并行的 Executors 数量，同时配合增加数据分片，整体上增加 CPU 的利用率，加快数据处理速度；若 Job 很容易发生内存溢出，就可以尝试增大分片数量，从而减少每个数据分片的大小，同时减少并行的 Executors 的数量，这样相同的内存资源分配给数量更少的 Executors，相当于增加了每个 Task 的内存分配，这样运行速度可能慢了些，但是总比 OOM 强；若数据量特别少，有大量的小文件生成，就可以尝试减少数据分片，没必要创建那么多的 Task，其实这种情况如果只是最原始的输入文件比较小，一般都能被注意到，但是如果是在运算过程中，比如应用某个 reduceByKey 或者 Filter 后，数据大量减少，这种低效情况就很少被留意到，要特别注意。

当为 Executors 分配好内存后，也可以通过参数进一步控制内存如何使用，如内存的多少可以用来计算，内存的多少可以用来做持久化操作等，涉及的参数在 Spark 的不同版本中有所不同，Spark 1. 6. 0 之前所使用的内存管理模式由类 StaticMemoryManager 实现，而 Spark 1. 6. 0 后所使用的内存管理模式由类 UnifiedMemoryManager 实现，进一步的调优细节请参考 9. 3. 5 内存调优。

需要注意的是，资源参数的调优不是一劳永逸的，需要根据实际作业执行情况动态调整。资源参数的调优没有一个固定的值，随着参数和配置的变化，性能的瓶颈是变化的，这就需要根据自己的实际情况（包括 Spark 作业中的 Shuffle 操作的数量、RDD 持久化操作的数量，以及 Spark WebUI 中显示的作业 GC 情况等）来动态调整配置。例如，在每台机器上部署的 Executors 数量增加时，性能一开始是增加的，同时也观察到 CPU 的平均使用率在增加；但是随着单台机器上的 Executors 数量越来越多，性能却开始下降了，因为随着 Executors 数量的增加，被分配到每个 Executors 的内存数量减小，在内存里直接进行的操作越来越少，Spill 到磁盘上的数据越来越多，自然性能就变差了。

9. 3. 4 序列化与压缩

序列化有时是 shuffle 和 cache 的瓶颈，合理地设置序列化，不但能提高 I/O 性能（包括

网络 I/O 和磁盘 I/O），还能减少内存的使用。

Spark 默认的序列化器是 org. apache. spark. serializer. JavaSerializer，也就是使用 ObjectOutputStream/ObjectInputStream API 来进行序列化和反序列化，但是这个默认的序列化器的性能和空间表现都比较差。Spark 同时支持使用 Kryo 序列化器 org. apache. spark. serializer. KryoSerializer，该序列化器更快，压缩率也更高。官方介绍，Kryo 序列化机制比 Java 序列化机制性能高了 10 倍左右，当然放到整个 Spark 程序中来考量，比重就没有那么大了，但是以 WordCount 为例，通常也很容易达到 30% 以上的性能提升。

Spark 之所以没有默认使用 Kryo 作为序列化器，是因为 Kryo 并不支持所有可序列化的类型，且要求最好注册所有需要进行序列化的自定义类型，这对于开发者而言略显麻烦。推荐在所有网络 I/O 密集型应用中使用 Kryo。事实上，Spark 对大多数常用的 scala 类都自动包含了 Kryo 序列化库。

在 Spark 中，以下地方会涉及序列化：在算子的函数中使用到外部变量时，该变量会被序列化后通过网络传输到 task 中；使用自定义的类型作为 RDD 的泛型类型时，所有的自定义类型对象都会进行序列化；使用需序列化的持久化策略时（比如 MEMORY_ONLY_SER），Spark 会将 RDD 中的每个 Partition 都序列化成一个大的字节数组；Spark Task 需要被序列化后从 Driver 发送到 Executor 上。这里涉及序列化的 4 个地方，前面 3 个使用的序列化器都可以通过设置 spark. serializer 来使用 Kryo 序列化，以提高性能；而 Spark Task 的序列化是通过 spark. closure. serializer 来配置的，但是目前它只支持 JavaSerializer。

使用 Kryo 序列化器时，只要设置序列化器为 Kryo，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为 RDD 泛型类型的自定义类型等）。事实上，如果没有注册自定义类型到 Kryo，Kryo 仍然可以工作，但需要对每个该类型的对象都存储完整的类名，当有百万条甚至更多的序列化记录时，这会额外占用更多的空间，所以推荐注册自定义的类到 Kryo，也可以通过设置 spark. kryo. registrationRequired 参数为 true 来强制注册，这样当 Kryo 遇到没有注册的类时，会抛出错误。

以下是使用 Kryo 序列化器的代码示例。

```
//创建 SparkConf 对象。
val conf = new SparkConf( )
//设置序列化器为 KryoSerializer。
conf. set( "spark. serializer" , "org. apache. spark. serializer. KryoSerializer" )
//设置需注册自定义类型到 Kryo
conf. set( "spark. kryo. registrationRequired" , "true" )
//注册要序列化的自定义类型
conf. registerKryoClasses( Array( classOf[ MyClass ] , classOf[ MyOtherClass ] ) )
```

在程序运行过程中，如果遇到了错误 NotSerializableException，就说明程序代码中使用了没有实现 Java 的 Serializable 序列化接口的类，需要修改相应的类，使其实现该接口。当由于程序中使用了很多类而不好判断究竟是哪个类引起该问题时，可以通过在 Spark – Submit 提交程序时，在 – driver – java – options 和 – executor – java – options 中指定 – Dsun. io. serialization. extended DebugInfo = true 来协助判断究竟是哪个类引起的该问题。

压缩和解压缩会消耗 CPU，但由于它能减少数据体积，使存储和传输更高效，所以在大数据分布式计算框架下很有用。Spark 中压缩相关参数为：①spark. rdd. compress，该参数

决定了 RDD 数据在序列化之后是否进一步进行压缩，然后再储存到内存或磁盘上，这个值默认是不压缩，但是如果在磁盘 I/O 的确成为问题或者 GC 问题真的没有其他更好的解决办法时，可以考虑启用 RDD 压缩；②spark. broadcast. compress，该参数决定了是否对 Broadcast 的数据进行压缩，默认值为 True，因为 Broadcast 的数据需要通过网络发送，而在 Executor 端又需要存储在本地 BlockMananger 中，所以通过压缩从而减小体积来减少网络传输开销和内存占用，通常都是有利于提高程序整体性能的；③spark. io. compression. codec，该参数决定了用来压缩内部数据，比如 RDD 分区、广播变量和 Shuffle 输出的数据等，所采用的压缩器，有 3 种选择：lz4、lzf 和 snappy，默认的是 Snappy，但和 Snappy 相比较，lzf 的压缩率比较高，故在有大量 Shuffle 的情况下，使用 lzf 可以提高 Shuffle 性能，进而提高程序整体效率。

9.3.5 内存调优

Spark 作为分布式计算框架，由于其优先使用内存，故对内存的管理对性能有重大影响，了解其内存管理机制及对内存的使用情况，也大大有助于程序的优化。

1. Spark 的内存管理机制

Spark 1.6.0 之前所使用的内存管理模式由类 StaticMemoryManager 实现（见图 9-18），而 Spark 1.6.0 之后所使用的内存管理模式由类 UnifiedMemoryManager 实现（见图 9-19），当然也可以通过参数 spark. memory. useLegacyMode 来配置使用哪种内存管理模式。

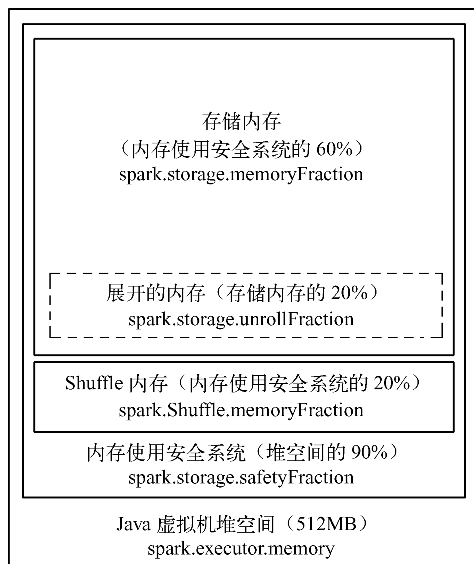


图 9-18　StaticMemoryManager

Executor 中对内存的使用涉及以下几点。

1）RDD 存储。当对 RDD 调用 Persist 或 Cache 方法时，RDD 的 Partition 会被存储到内存中。

2）Shuffle 操作。Shuffle 时，需要缓冲区来存储 Shuffle 的输出和聚合的中间结果。

3）用户代码。用户编写的代码所能使用的内存空间是整个堆空间除了上述两点之后剩下的空间。

StaticMemoryManage 模式下，堆空间分为 Storage 区和 Shuffle 区。Storage 区所能使用的堆空间的比例由 spark. storage. memoryFraction 指定，默认值为 0.6，为了避免内存溢出的风

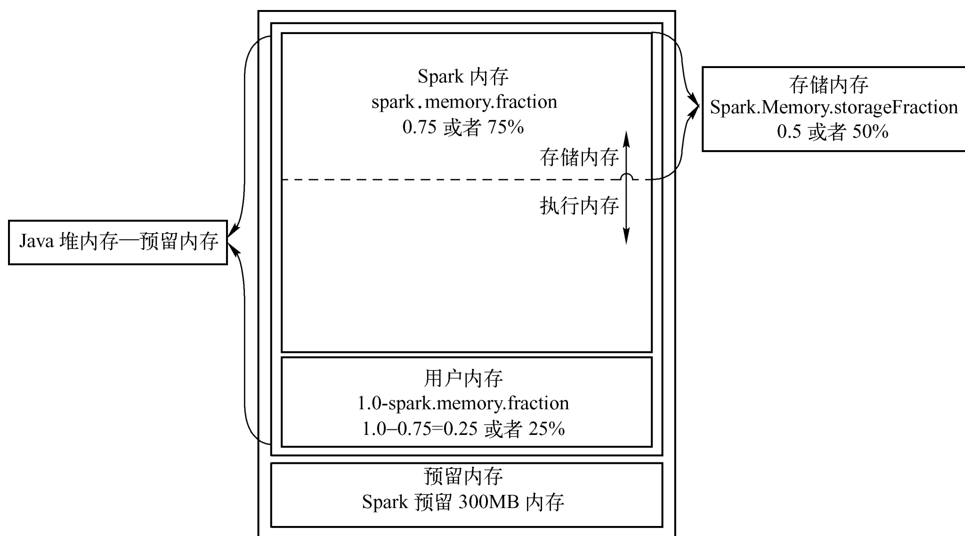


图 9-19 UnifiedMemoryManager

险，使用参数 spark. storage. safetyFraction 来指定安全区比例，该参数的默认值为 0.9，故实际可用的 Storage 区为堆空间的 0.54（0.9×0.6=0.54）；shuffle 区所能使用的堆空间的比例由 spark. shuffle. memoryFraction 指定，默认值为 0.2，为了避免内存溢出的风险，使用参数 spark. shuffle. safetyFraction 来指定安全区比例，该参数默认值为 0.8，故实际可用的 shuffle 区为堆空间的 0.16（0.8×0.2=0.16）。如果 Spark 作业中有较多的 RDD 持久化操作，可以将 spark. storage. memoryFraction 的值适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果 Spark 作业中的 Shuffle 类操作比较多，而持久化操作比较少，那么可以将 spark. storage. memoryFraction 的值适当降低一些，而将 spark. shuffle. memoryFraction 的值适当提高一些，以避免 shuffle 过程中数据过多时内存不够用，必须溢写到磁盘上而降低性能；此外，如果发现作业由于频繁的 GC 导致运行缓慢（通过 spark web ui 可以观察到作业的 GC 耗时），意味着 task 执行用户代码的内存不够用，那么同样建议调低参数 spark. storage. memoryFraction 和 spark. shuffle. memoryFraction 的值。

UnifiedMemoryManager 模式下，整个堆空间分为 Spark Memory 和 User Memory，在 Spark Memory 内部又分为 Storage Memory 和 Execution Memory，Storage Memory 和 Execution Memory 并没有硬界限，可以相互借用空间。可以通过参数 spark. memory. fraction（默认为 0.75）来设置 Spark Memory 所占的整个堆空间的比例，剩下的空间就是 User Memory（默认为 1－0.75＝0.25）；通过 spark. memory. storageFraction（默认为 0.5）设置 Storage Memory 所占的 Spark Memory 的比例。根据实际程序中 Cache 的多少、Shuffle 的多少和对象的多少等，可以调整上述各个参数来调整各个内存区的大小，进而优化程序。

下面重点介绍 UnifiedMemoryManager 模式下的相关参数。

1）spark. storage. memoryFraction。参数说明：该参数用于设置 RDD 持久化数据在 Executor 内存中所占的比例，默认为 0.6。也就是说，默认 Executor 60% 的内存，可以用来保存持久化的 RDD 数据。根据不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。参数调优建议：如果 Spark 作业中有较多的 RDD 持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中，避免内存不够缓存所有的

数据，导致数据只能写入磁盘中，降低了性能。但是如果 Spark 作业中的 Shuffle 类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的 GC 导致运行缓慢（通过 Spark web UI 可以观察到作业的 GC 耗时），意味着 task 执行用户代码的内存不够用，那么同样建议调低这个参数的值。

2）spark. shuffle. memoryFraction。参数说明：该参数用于设置 Shuffle 过程中一个 Task 拉取到上个 Stage 的 Task 的输出后，进行聚合操作时能够使用的 Executor 内存的比例，默认为 0. 2。也就是说，Executor 默认只有 20% 的内存用来进行该操作。Shuffle 操作在进行聚合时，如果发现使用的内存超出了这个 20% 的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。参数调优建议：当 Spark 作业中的 RDD 持久化操作较少、Shuffle 操作较多时，建议降低持久化操作的内存占比，提高 Shuffle 操作的内存占比比例，避免 Shuffle 过程中数据过多时内存不够用，必须溢写到磁盘上，从而降低了性能。此外，如果发现作业由于频繁的 GC 导致运行缓慢，意味着 Task 执行用户代码的内存不够用，那么同样建议调低这个参数的值。

2. 确定内存消耗

可以在程序中通过 Cache 方法将 RDD Cache 到内存中，然后通过 WebUI 的 Storage 页面查看该 RDD 占用了多少内存，或查看 Driver 的日志。

有很多工具也可以帮助用户了解内存的消耗，比如 JVM 自带的内存消耗诊断工具，如 JMap、JSonsole 等；第三方工具如 IBM JVM Profile Tools 等。

通过这些方法，可以了解各种数据结构、广播变量等对内存的占用，从而选择使用对内存更为友好的数据结构。

3. 数据结构调优

详情请参阅 9. 3. 1 节的“准则八：使用优化的数据结构”。

4. 序列化要持久化的 RDD

详情请参阅 9. 3. 4 节。

5. 垃圾回收的调优

由于 Spark 运行在 JVM 平台上，而垃圾回收是 JVM 的“死穴”，所以内存的调优肯定离不开垃圾回收的优化，更详细的信息请参考 9. 3. 9 节。

9. 3. 6 广播大变量

有时在开发过程中，会遇到需要在算子函数中使用外部大变量的场景（如 100 MB 的大集合），那么此时就可以考虑使用 Spark 的广播（Broadcast）功能来提升性能。

Spark 使用了 Shared – Nothing 架构，数据以分区的形式散落在各个结点上，每个结点都有自己的 CPU、内存和存储资源。tasks 并没有共享的全局内存区域，Driver 和 Task 通过通信来共享数据。比如，当一个 RDD 算子所使用的函数中引用了一个来自 Driver 的变量时，Spark 会将该变量的一个副本随一个 Task 一起发送给 Executors，然后每个 Task 得到一个该变量的副本，并以只读的形式访问它，任何对该变量的修改都是本地的，并不会返回给 Driver。这个发送动作在每个 Stage 的开始都会发生一次。

这种默认的行为在 Driver 和 Task 共享较大的变量（如静态查询表），且 Job 有多个 Stage 时并不高效，比如该静态查询表的大小是 100 MB 且该 Job 有 10 个 Stage，则 Spark 会将这 100 MB 的数据发送给每个 Executor 共 10 次。大量的变量副本在网络中传输的性能开销，以

及在各个结点的 Executor 中占用过多内存导致的频繁 GC，都会极大地影响性能。这显然并不高效，Spark 为此推出了广播变量。

Spark 的广播变量只会发送给 Executor 结点一次，且发送后广播变量会以非序列化的形式保存在 Executor 内存中，而不会为每个 Task 都发送一份。Spark 还会使用高效的广播算法（Http 或 Torrent 方式）来分发变量，所以网络通信的开销并不大。熟悉源代码的朋友都知道，在 Spark 的 HadoopRDD 中，就采用了广播来进行 Hadoop 的 JobConf 的传输以提高效率。广播后的变量会保证每个 Executor 的内存中只驻留一份该变量的副本，而 Executor 中的 task 执行时共享该 Executor 中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对 Executor 内存的占用开销，降低 GC 的频率。

需要说明的是，由于每个 Stage 的 Task 公用的数据在 Stage 开始时都会向每个 Executor 结点发送一次，发送后这些公用数据会以序列化的形式缓存在 Executor 中，然后各个任务在运行时反序列化这些公用的数据，以获得一个只读的副本供自己使用，这就意味着显式创建广播变量的模式，只有在 Job 有多个 Stage，且多个 Stage 的 Task 需要访问来自 Driver 的相同数据时，或者需要以非序列化的形式缓存数据时，才真正有意义。

事实上，Spark 会在 Master 上打印出每个任务序列化后的大小，通常来讲，大于 20 KB 的任务就可以考虑是否可以通过广播机制进行优化。

可以通过在一个只读变量 v 上调用 SparkContext. broadcast(v) 来创建广播变量，广播变量本质上是围绕着变量 v 的封装，广播后可以通过 value 方法访问这个广播变量的值。在创建了广播变量之后，在集群上的所有函数中都应该使用它来替代使用 v，这样 v 就不会不止一次地在结点之间传输了。另外，为了确保所有的结点获得相同的变量，变量 v 在被广播之后就不应该再修改。

实例代码如下。

```
//以下代码在算子函数中使用了外部的变量
//此时没有做任何特殊操作,每个 Stage 开始时都会向每个 Executor 结点发送一次该变量
val V = …
rdd1. map(V…)
//以下代码将 V 封装成了 Broadcast 类型的广播变量
//在算子函数中,使用广播变量时,首先会判断当前 Task 所在的 Executor 内存中是否有变
  量副本。如果有则直接使用;如果没有则从 Driver 或者其他 Executor 结点上远程拉取一
  份放到本地 Executor 内存中,每个 Executor 内存中只会驻留一份广播变量副本
val V = …
val V Broadcast = sc. broadcast(V)
rdd1. map(V Broadcast. value…)
```

另外一个使用广播变量的常用场景是，当对两个表进行 Join 操作时，为了避免 Shuffle，经常将较小的表广播到 Executor 上。

9.3.7 持久化与 checkpoint

因为 Spark 程序执行的特性，即延迟执行和基于 Lineage 最大化的 pipeline，当 Spark 中由于对某个 RDD 的 Action 操作触发了作业时，会基于 Lineage 从后往前推，找到该 RDD 的源头 RDD，然后从前往后计算出结果。很明显，如果对某个 RDD 执行了多次 Transformation

和 Action 操作，每次 Action 操作触发了作业时都会重新从源头 RDD 处计算一遍来获得该 RDD，然后再对这个 RDD 执行相应的操作。当 RDD 本身计算特别复杂和耗时（如某个计算时长超过半个小时）时，这种方式的性能显然是很差的，此时必须考虑对计算结果的数据进行持久化。

持久化就是将计算出来的 RDD 根据配置的持久化级别，保存到内存或磁盘中，以后每次对该 RDD 进行算子操作时，都会直接从内存或磁盘中提取持久化的 RDD 数据，然后执行算子，而不会从源头处重新计算一遍这个 RDD，再执行算子操作，这无疑会提高程序执行的效率。

建议对多次使用的 RDD，计算特别复杂和耗时，或计算链条特别长的 RDD，进行持久化。只需要对希望缓存的 RDD Persist 或 Cache 方法进行标记，这两种方法的区别在于，Cache 是将 RDD 持久化到内存里，而 Persist 可以指定不同的持久化级别。在标记之后，当由 Action 触发 Job 导致了该 RDD 的计算后，计算结果就会根据指定的持久化级别被持久化到内存或磁盘中。

RDD 的各个持久化级别如表 9-1 所示。

表 9-1　RDD 的持久化级别

持久化级别	含义解释
MEMORY_ONLY	使用未序列化的 Java 对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据就可能有部分不会进行持久化。那么下次对这个 RDD 执行算子操作时，那些没有被持久化的数据需要从源头处重新计算一遍。这是默认的持久化策略，使用 cache( ) 方法时，实际使用的就是这种持久化策略
MEMORY_AND_DISK	使用未序列化的 Java 对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个 RDD 执行算子时，持久化在内存和磁盘文件中的数据会被读取出来使用
MEMORY_ONLY_SER	基本含义同 MEMORY_ONLY。唯一的区别是，会将 RDD 中的数据进行序列化，RDD 的每个 partition 会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁 GC
MEMORY_AND_DISK_SER	基本含义同 MEMORY_AND_DISK。唯一的区别是，会将 RDD 中的数据进行序列化，RDD 的每个 partition 会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁 GC
DISK_ONLY	使用未序列化的 Java 对象格式，将数据全部写入磁盘文件中
OFF_HEAP	将 RDD 以序列化的方式存储在 Alluxio 中（曾用名 Tachyon），而不是 Executor 的内存中，减少了垃圾回收的压力 当 RDD 存储在 Alluxio 上时，Executors 的崩溃不会造成 RDD 数据的丢失 事实上，Alluxio 带来的不止这些，它是一套基于内存的分布式文件存储系统，为跨程序、跨框架地共享内存数据提供了一套方案
MEMORY_ONLY_2， MEMORY_ONLY_SER_2， MEMORY_AND_DISK_2， MEMORY_AND_DISK_SER_2， DISK_ONLY_2， 等等.	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据都复制一份副本，并将副本保存到其他结点上。这种基于副本的持久化机制主要用于进行容错。假如某个结点挂掉，结点的内存或磁盘中的持久化数据丢失了，那么后续对 RDD 计算时还可以使用该数据在其他结点上的副本。如果没有副本，就只能将这些数据从源头处重新计算一遍了

Spark 提供了不同的持久化级别，以满足内存使用率和 CPU 效率的均衡。如何选择合适的持久化级别呢？

通常遵循的准则是，优先考虑内存，内存放不下就考虑序列化后放在内存里，尽量不要存储到磁盘上，因为一般 RDD 的重新计算比从磁盘中读取更快，只有在需要更快的恢复时才使用备份级别（所有的存储级别都可以通过重新计算来提供全面的容错性，但是备份级别允许用户继续在 RDD 的备份上执行任务，而无须重新计算丢失的分区）。

按照顺序来讲，通常按照以下方式来选择。

采用默认情况下性能最高的 MEMORY_ONLY。该持久化级别下，不需要进行序列化与反序列化操作，就避免了这部分的性能开销；对这个 RDD 的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他结点上。但是这里必须要注意的是，使用该持久化级别的前提是，集群的内存必须足够大，可以绰绰有余地存放整个 RDD 的所有数据，如果 RDD 中的数据比较多（比如几十亿条），直接用这种持久化级别可能会导致 JVM 的 OOM 内存溢出异常。

如果使用 MEMORY_ONLY 级别时发生了内存溢出，那么建议尝试使用 MEMORY_ONLY_SER 级别。该级别会将 RDD 数据序列化后再保存到内存中，此时每个 partition 仅仅是一个字节数组而已，大大减少了对象数量，并降低了内存占用。这种级别比 MEMORY_ONLY 多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果 RDD 中的数据量过多，还是可能会导致 OOM 内存溢出的异常。

如果纯内存的级别都无法使用，那么建议使用 MEMORY_AND_DISK_SER 策略，而不是 MEMORY_AND_DISK 策略。因为此时 RDD 的数据量很大，内存无法完全放下，需要通过序列化来节省内存和磁盘的空间开销。该策略会优先尽量尝试将数据缓存到内存中，内存缓存不下时才会写入磁盘。

通常不建议使用 DISK_ONLY 级别。因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次该 RDD。

通常不建议使用后缀为_2 的备份级别。因为该级别必须将所有的数据都复制一份副本，并发送到其他结点上，而数据复制和网络传输会导致较大的性能开销，除非是作业的高可用性要求很高，否则不建议使用。

这里需要重点指出的是，推荐尝试使用 OFF_HEAP 方式，因为该方式将 RDD 数据持久化到了 Alluxio 中（曾用名 Tachyon），而不是 Executor 的内存中。OFF_HEAP 方式有以下几个优势：允许多个 Executors 共享同一个内存池；显著地减少了垃圾回收的开销；如果某个 Executors 崩溃，缓存的数据不会丢失。事实上，Alluxio 带来的不止这些，它是一套基于内存的分布式文件存储系统，为跨程序、跨框架地共享内存数据提供了一套方案。更详细的情况请查阅 Alluxio 官网。

需要说明的是，持久化会最大化地保留整个 RDD 的所有分区数据，但是如果当前的计算需要内存空间而空闲内存不够，那么持久化在内存中的数据必须让出空间，此时如果 RDD 的持久化级别指定了可以把数据放在磁盘上，那么部分分区数据就可以从内存转入磁盘，否则数据就会丢失；Spark 的持久化也具有容错性，当持久化的 RDD 的某一分区丢失后，后续计算该分区后也会自动重新计算并持久化该分区；同时，Spark 也有自己的一套机制，自动监控每个结点上的缓存使用率，并通过 LRU（Least Recently Used，近期最少使用）算法删除过时的缓存数据（当然也可以手动使用 RDD. unpersist( )方法来删除）。也就是说，

持久化的数据在执行时不一定被有效地持久化了，要想知道要持久化的 RDD 有没有被正确地持久化，可以通过查看 WebUI 的 Executor 页面，详见 9.2.8 节。

下面是一个使用 Persist 的示例。

```
//cache 使用示例:
val rdd1 = sc. textFile("hdfs://master:9000/data/README. md "). cache()
rdd1. map(…)
rdd1. reduce(…)
// persist()方法表示:手动选择持久化级别,并使用指定的方式进行持久化
//例如,StorageLevel. MEMORY_AND_DISK_SER 表示,内存充足时优先持久化到内存中,内
  存不充足时持久化到磁盘文件中。而且其中的_SER 后缀表示使用序列化的方式来保存
  RDD 数据,此时 RDD 中的每个 partition 都会序列化成一个大的字节数组,然后再持久化
  到内存或磁盘中。序列化的方式可以减少持久化的数据对内存/磁盘的占用量,进而避
  免内存被持久化数据占用过多,从而发生频繁 GC
// persist 使用示例:
val rdd1 = sc. textFile("hdfs://master:9000/data/README. md "). persist(StorageLevel. MEMORY_
AND_DISK_SER)
rdd1. map(…)
rdd1. reduce(…)
```

把数据通过 Persist 或 Cache 持久化到内存或磁盘中，虽然是快速的但却不是最可靠的，Checkpoint 机制的产生就是为了更加可靠地持久化数据以复用 RDD 计算数据，通常针对整个 RDD 计算链条中特别需要数据持久化的环节，启用 Checkpoint 机制来确保高容错和高可用性。

可以通过调用 SparkContext. setCheckpointDir 方法来指定 Checkpoint 时持久化的 RDD 的数据的存放位置。在 Checkpoint 中可以指定把数据以多副本的方式存放在本地或 HDFS 中（在生产环境下通常是放在 HDFS 中，从而天然地借助 HDFS 高容错和高可靠的特征完成最大化的可靠的持久化）；同时为了提高效率，可以指定多个目录。

需要说明的是，Checkpoint 同 Persist 一样是惰性执行的，在对某个 RDD 标记了需要 Checkpoint 后，并不会立即执行，只有在后续有 Action 触发 Job 从而导致了该 RDD 的计算，且在这个 Job 执行完成后，才会从后往前回溯找到标记了 Checkpoint 的 RDD，然后重新启动一个 Job 来执行具体的 Checkpoint，所以一般都会对需要进行 Checkpoint 的 RDD 先进行 Persist 标记，从而把该 RDD 的计算结果持久化到内存或者磁盘上，以备 checkpoint 复用。

下面是一个使用 Checkpoint 的示例。

```
// checkpoint 使用示例:
//配置 CheckpointDir
sc. setCheckpointDir("hdfs://master:9000/spark/checkpoint)
val rdd1 = sc. textFile("hdfs://master:9000/data/README. md "). cache()
//对 rdd1 标记 checkpoint
rdd1. checkpoint()
//action 触发了 Job 才能导致 checkpoint 的真正执行
rdd1. count()
```

9.3.8 数据本地性

数据本地性是指数据与执行的代码的远近程度。基于数据与执行的代码各自所在的位置，从近到远的本地性级别依次如下。

- PROCESS_LOCAL：数据与执行的代码在同一个 JVM 内存中，数据已经缓存到执行代码所在的 Executor 的内存中了，即读取缓存在本地结点的数据，这是效率最高的数据本地性级别（该本地性级别和 cache 有关）。
- NODE_LOCAL：数据与执行的代码在同一个结点上，需要从执行代码所在的结点上的本地存储中读取出数据，即读取本地结点硬盘数据。比如，数据在同一个结点的 hdfs 上，或在同一个结点的另一个 Executor 上。因为数据需要在进程间移动，该级别比 PROCESS_LOCAL 稍慢。
- NO_PREF：从任意地方访问该数据的速度都一样，没有具体的位置偏好。
- RACK_LOCAL：数据所在的 Server 与执行代码的 Server 在同一台机架上，一般需要通过一台交换机在网络上传输到执行代码所在的结点上。
- ANY：数据位于不在同一台机架的其他集群的结点上，即读取非本地结点数据。

数据本地性对程序的性能有很大的影响。如果数据和要执行的代码在一起，计算速度就很快；而当它们不在一起时，其中一个就必须移动到另一个所在的位置，而数据在网络上的传输会导致大量的延时和开销，毕竟磁盘 I/O 和网络 I/O 都是耗时的操作。

在大数据处理模式下，一般代码都比数据小，所以序列化了的代码的移动比数据的移动更快，分布式计算系统的精髓就在于移动代码而非移动数据，即“数据不动代码动”。数据本地性的原则就是尽量避免数据在网络上甚至磁盘上的传输，尽量将计算移到数据所在的结点上进行。Spark 的调度机制就是基于这一准则来进行调度的。

Spark 中任务的处理需要考虑数据本地性的场合，基本上就两种：一是数据来源于外部数据源（如 HDFS）；二是数据来源于 RDD Cache（即由 CacheManager 从 BlockManager 中读取，或者 Streaming 数据源 RDD）。其他情况下，不涉及 Shuffle 操作的不构成划分 Stage 和 Task 的基准，也就不存在本地性问题；而如果是 ShuffleRDD，由于其本地性始终为 No Prefer，因此其实也不存在本地性问题。

在理想情况下，任务当然是分配在可以从本地读取数据的结点上时（同一个 JVM 内部或同一台物理机器内部）的运行时性能最佳。但是每个任务的执行速度无法准确估计，所以很难在事先获得全局最优的执行策略。当 Spark 应用程序得到一个计算资源时，如果没有可以满足最佳本地性需求的任务可以运行，即任何空闲的 Executor 上都没有尚未处理的数据时，计算框架通常有两种选择：①一直等待直到待处理的数据所在的结点的 CPU 空闲下来，然后调度处理该批数据的 Task 到该结点上，这样能更好地匹配任务的本地性；②不进行等待，直接将处理该批数据的 Task 调度到其他结点上执行，当然此时该批数据需要移动到相应的结点上，这是退而求其次，运行一个本地性条件稍差一点的任务。

Spark 所遵循的原则是，调度作业时优先调度到有最好本地性的结点上去执行 Task，当较高的数据本地性级别不能满足时，Spark 退而求其次调度作业到次好的本地性结点。具体实施时，Spark 混合了这两种方案，即会首先等待一小会儿以期望忙碌的 CPU 空闲下来，如

果这个等待的“一小会儿”到了而 CPU 仍然忙碌，就会移动数据到有空闲 CPU 的结点去执行。这个需要等待的“一小会儿”是可以通过参数 spark. locality. wait、spark. locality. wait. process、spark. locality. wait. node 和 spark. locality. wait. rack 来配置的，这几个参数一起决定了 Spark 任务调度模块在得到待分配任务时，如果没有更好的本地性级别，暂时不分配任务而是等待一小会儿以期望获得更好的本地性级别的等待时间。

- spark. locality. wait：调度作业时，如果没有更好的本地性级别，退而求其次调度到次好的本地性级别的等待时间，默认是 3s。该时间是所有本地性级别的等待时间（process – local、node – local、rack – local 和 any）的默认值。当然也可以分别配置各个本地性级别的等待时间。
- spark. locality. wait. process：PROCESS_LOCAL 本地性级别的等待时间，即尝试访问缓存在 Executor 进程的内存中数据的等待时间，默认值是 spark. locality. wait 的值。
- spark. locality. wait. node：NODE_LOCAL 本地性级别的等待时间，默认值是 spark. locality. wait 的值。比如，可以将该值设置为 0，以跳过 NODE_LOCAL 本地性级别，而直接尝试 RACK_LOCAL。
- spark. locality. wait. rack：RACK_LOCAL 本地性级别的等待时间，默认值是 spark. locality. wait 的值。

一般来说，这几个参数默认的配置均不需要修改，但如果 Task 执行时间长且本地性级别差（可以通过 WebUI 观察到），可以调高这些参数，使系统等待更长的时间，以满足更好的本地性来运行作业。

需要指出的是，在应用程序刚启动，处理提交的第一批任务时，由于当作业调度模块开始工作时，具体处理任务的 Executor 可能还没有完全注册完毕，因此一部分任务会被错误地放置到 No Prefer 的队列中，由于这部分任务的优先级仅次于数据本地性满足 Process 级别的任务，它们就有可能被优先分配到非本地结点上执行（在的确没有 Executors 在对应的结点上运行，或者的确是 No Prefer 的任务（如 shuffleRDD）的情况下，将任务放置到 No Prefer 的队列中确实是比较优化的选择，但是在这种情况下，由于只是这部分 Executor 还没来得及注册上而已，这种方法不是一个好选择）。这种情况下，即使调大这几个参数的数值也没有帮助。针对这种情况，Spark 有一些已经完成的和正在进行中的 release patch（修复方案），试图通过动态调整 No Prefer 队列、监控 executors 注册比例等方式来给出更加智能的解决方案。然而，也可以根据自身集群的启动情况，通过在创建 SparkContext 之后，主动 Sleep 几秒的方式来简单地解决这个问题。

9.3.9 垃圾回收调优

与 Hadoop、HBase 生态圈的众多项目等一样，Spark 的运行离不开 JVM 的支持，同时由于 Spark 立足于内存计算，常常需要在内存中存放大量数据，再加上 Spark 同时支持批处理和流式处理，对于程序吞吐量和延迟都有较高要求，所以 Spark 对于 JVM 垃圾回收机制（GC）的依赖更加突出。而垃圾回收是 JVM 的“死穴”，它会影响程序的性能甚至还会带来系统卡顿，所以 GC 参数的调优在 Spark 应用实践中显得尤为重要。

首先需要获取一些统计信息，包括内存回收的频率、内存回收耗费的时间等。为了获取这些统计信息，可以把参数 – verbose:gc – XX:+PrintGCDetails – XX:+PrintGCTimeStamps 添加到

环境变量 SPARK_JAVA_OPTS 中。设置完成后，Spark 作业运行时，可以在日志中看到每一次垃圾回收的详细情况。需要注意的是，这些信息保存在集群的 Executors 中而不是 Driver 中。

Spark 垃圾回收调优的目标是确保只有长时间存活的 RDD 才保存到老生代区域；同时，新生代区域需要足够大以保存生命周期比较短的对象。这样，在任务执行期间可以避免执行 full GC。下面是一些可能有用的执行步骤。

通过收集 GC 信息检查内存回收是否过于频繁，如果在任务结束之前执行了很多次 full GC，则表明任务执行的内存空间不足；在打印的内存回收信息中，如果老生代接近消耗殆尽，那么减少用于缓存的内存空间，这可以通过配置属性 spark. storage. memoryFraction 来完成，通过减少缓存对象来提高执行速度是非常值得的；如果有过多的 minor GC 而不是 full GC，那么为 Eden 分配更大的内存是有益的，可以为 Eden 分配大于任务执行所需要的内存空间。如果 Eden 的大小确定为 E，那么可以通过 $-Xmn = 4/3 \times E$ 来设置新生代的大小（将内存扩大到 4/3 是考虑到 survivor 所需要的空间）。举一个例子，如果任务从 HDFS 读取数据，那么任务需要的内存空间可以从读取的 block 数量估算出来。注意解压后的 blcok 通常为解压前的 2 ～ 3 倍。所以，如果需要同时执行 3 或 4 个任务，block 的大小为 64 M，可以估算出 Eden 的大小为 $4 \times 3 \times 64$ MB；监控内存回收的频率及消耗的时间，并修改相应的参数设置。

9.3.10 Shuffle 调优

在 Spark 程序中，Shuffle 是性能的最大瓶颈，因为 Shuffle 的过程往往伴随着磁盘 I/O 与网络 I/O 等开销，程序编写的一个准则就是“尽量避免使用需要 Shuffle 的算子，且在必须 Shuffle 时尽量减少 Shuffle 的数据量”，而在 Shuffle 不可避免时，也要尽量优化 Shuffle 的方方面面以调优性能。

这里针对 Shffule 过程中的一些主要参数，详细讲解各个参数的功能、默认值，以及基于实践经验给出的调优建议。更详细的关于 Shuffle 的解析，请参阅本书第 7 章 Shuffle 机制。

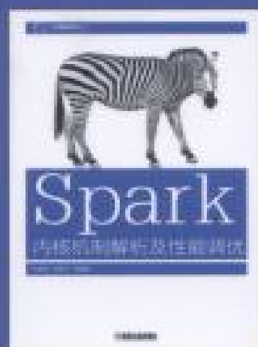
- spark. shuffle. file. buffer：该参数用于设置 shuffle write task 的 BufferedOutputStream 的 buffer 缓冲大小。将数据写到磁盘文件之前，会先写入 Buffer 缓冲中，待缓冲写满之后，才会溢写到磁盘，默认值为 32 KB。调优建议：如果作业可用的内存资源较为充足，可以适当增加这个参数的大小（比如 128 KB），从而减少 shuffle write 过程中溢写磁盘文件的次数，也就可以减少磁盘 I/O 次数，进而提升性能。
- spark. reducer. maxSizeInFlight：该参数用于设置 Shuffle Read Task 的 Buffer 缓冲大小，而这个 Buffer 缓冲决定了每次能够拉取多少数据，默认值为 48 MB。调优建议：如果作业可用的内存资源较为充足，可以适当增加这个参数的大小（如 96 MB），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。
- spark. shuffle. io. maxRetries：shuffle read task 从 shuffle write task 所在结点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的，该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。默认值为 3。调优建议：对于那些包含了特别耗时的 Shuffle 操作的作业，建议增加重试最大次数（如 30 次），以避免由于 JVM 的 Full GC 或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿～上百亿）的 Shuffle 过程，调节该参数可以大幅度提升稳定性。

- spark. shuffle. io. retryWait：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认值是 5 s。调优建议：建议加大间隔时长（如 30 s），以增加 Shuffle 操作的稳定性。
- spark. shuffle. memoryFraction 与 spark. storage. memoryFraction （StaticMemoryManager 模式下），或 spark. memory. fraction 与 spark. memory. storageFraction （UnifiedMemoryManager 模式下）：详情参考 9. 3. 3 资源参数调优一节。
- spark. shuffle. manager：该参数用于设置 ShuffleManager 的类型。Spark 1. 5 以后，有 3 个可选项：hash、sort 和 tungsten – sort。HashShuffleManager 是 Spark 1. 2 以前的默认选项，但是 Spark 1. 2 及之后的版本默认都是 SortShuffleManager。Tungsten – Sort 与 Sort 类似，但是使用了 Tungsten 计划中的堆外内存管理机制，内存使用效率更高。调优建议：由于 SortShuffleManager 默认会对数据进行排序，因此如果用户的业务逻辑中需要该排序机制，则使用默认的 SortShuffleManager 即可；而如果用户的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过 bypass 机制或优化的 HashShuffleManager 来避免排序操作，同时提供较好的磁盘读写性能。这里需要注意的是，随着 Tungsten – Sort 越来越稳定，可以尝试使用 Tungsten – Sort。
- spark. shuffle. sort. bypassMergeThreshold：当 ShuffleManager 为 SortShuffleManager 时，如果 Shuffle Read Task 的数量小于这个阈值，则 Shuffle Write 过程中不会进行排序操作，而是直接按照未经优化的 HashShuffleManager 的方式去写数据，但是最后会将每个 Task 产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件，默认值是 200。调优建议：当使用 SortShuffleManager 时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于 Shuffle Read Task 的数量。那么此时就会自动启用 bypass 机制，此时 Map 端就不会进行排序了，从而减少了排序的性能开销。但是这种方式下依然会产生大量的磁盘文件，因此 Shuffle Write 性能有待提高。
- spark. shuffle. consolidateFiles：如果使用 HashShuffleManager，该参数有效。如果设置为 true，那么就会开启 Consolidate 机制，会大幅度合并 Shuffle Write 的输出文件，在 Shuffle Read Task 数量特别多的情况下，这种方法可以极大地减少磁盘 I/O 开销，提升性能。默认值是 False。调优建议：如果的确不需要 SortShuffleManager 的排序机制，那么除了使用 bypass 机制外，还可以尝试将 spark. shffle. manager 参数手动指定为 hash，使用 HashShuffleManager，同时开启 consolidate 机制。
- spark. local. dir：Shuffle 的 Write 阶段使用的本地磁盘目录。当 Shuffle 阶段磁盘 I/O 时间过长时，可以将该参数设置为多个 I/O 速度快的磁盘，通过增加 I/O 来优化 Shuffle 性能。

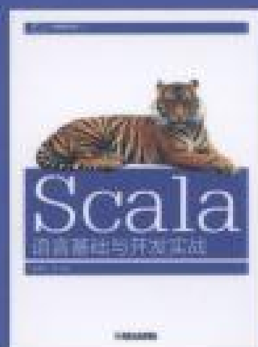
Section 9. 4 小结

本章首先讲述了 Spark 的配置机制，以帮助我们了解如何配置程序的运行参数；然后讲解了如何进行性能诊断，以了解性能状况找出性能瓶颈；最后详细描述了各类常见的性能优化的原理与方法。需要注意的是，性能调优不是一劳永逸的，随着集群状况和作业执行情况的变化，性能的瓶颈也是变化的，这就需要我们分析诊断动态变化的情况，基于对 Spark 内核机制的理解，灵活动态地调整，来达到优化的效果。

大数据科学丛书
BIG DATA SCIENCE



本书从源码分析入手，详细讲述Spark RDD的功能以及内部实现机制、Spark的集群部署、内部调度、Executor、Shuffle的内部机制，进而讲述“钨丝计划”，并特别介绍Spark性能调优的内容。







Scala语言基础与开发实战
ISBN：978-7-111-54169-1
定价：69.00元

Spark原理、机制及应用
ISBN：978-7-111-52928-6
定价：49.00元

Spark核心源码分析与开发实战
ISBN：978-7-111-52860-9
定价：75.00元

作者沟通交流方式：
微信公众号：DT_Spark
微信号：18610086859
新浪微博：@lovepains

书籍QQ交流群：
DT大数据梦工厂①：462923555
DT大数据梦工厂②：437123764
DT大数据梦工厂③：418110145

欲获取更多资源及图书信息请关注





机械工业出版社
微信服务号

计算机分社
微信订阅号

机工互联网家
微信公众号

为中华崛起传播智慧

地址：北京市百万庄大街22号
邮政编码：100037

电话服务
服务咨询热线：010-88361066
读者购书热线：010-68326294
010-88379203

网络服务
机工官网：www.cmpbook.com
机工官博：weibo.com/cmp1952
金书网：www.golden-book.com
教育服务网：www.cmpedu.com
封面无防伪标均为盗版

51CTO.com
技术成就梦想

上架指导 计算机/大数据

策划编辑◎王 斌 / 封面设计◎MXK

ISBN 978-7-111-55442-4



定价：59.00元